June 1977

This document describes the fundamentals of the FORTRAN IV and FORTRAN IV-PLUS language elements, as implemented for the PDP-11 systems. Shaded text pertains only to FORTRAN IV-PLUS features. Information pertaining to VIRTUAL arrays applies only to FORTRAN IV.

PDP-11
FORTRAN
Language Reference Manual

Order No. DEC-11-LFLRA-C-D
Including DEC-11-LFLRA-C-DN1

SUPERSESSION/UPDATE INFORMATION: This manual contains information concerning FORTRAN IV V02 and FORTRAN IV-PLUS V02 as of June 1977.

OPERATING SYSTEM AND VERSION:
RSX-11M V3
RSX-11D V6.2
RSTS/E V6B
IAS V2
RT-11 V3

SOFTWARE VERSION:
FORTRAN IV V2
FORTRAN IV-PLUS V2.5

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754.

digital equipment corporation · maynard, massachusetts

First Printing, June 1974
Revised: December 1974
December 1975
June 1977

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECSYSTEM-20	TYPESET-11

CONTENTS

FIGURES

TABLES

PREFACE

FORTRAN is a problem oriented language designed to permit scientists and engineers to express mathematical operations in a form with which they are familiar. It is also used in a variety of applications including process control, information retrieval, and commercial data processing.

This document describes the form of the basic elements of the FORTRAN program, the FORTRAN statements. The document is a reference manual, and, while it can well be used by an inexperienced FORTRAN programmer, it is not designed to function as a tutorial manual.

Because this document serves as the FORTRAN Language Reference Manual for several of the operating systems which run on the PDP-11 family of computers, it makes no reference to system dependent information. Associated with this document, however, should be the FORTRAN User's Guide containing the necessary information for running a FORTRAN program on a specific operating system.

DOCUMENTATION CONVENTIONS

Throughout this manual the following notations are used to denote special non-printing characters:

→|	Tab character (TAB key or CTRL/I key combination)
Δ (delta)	Space character (SPACE bar)

This document serves as the FORTRAN language reference manual for two PDP-11 FORTRAN processors: FORTRAN IV and FORTRAN IV-PLUS.

FORTRAN IV-PLUS is a superset of FORTRAN IV, with one exception: the VIRTUAL array feature, described in Appendix D and mentioned at several points in the main portion of the manual, pertains only to FORTRAN IV.

Language elements common to both processors are presented without background shading. Language elements available only in FORTRAN IV-PLUS are printed against a shaded background.

SYNTAX NOTATION

The following conventions are used in the description of FORTRAN statement syntax.

1. Upper case words and letters, as well as punctuation marks other than those described in this section, are written as shown.

2. Lower case words indicate that a value is to be substituted. The accompanying text specifies the nature of the item to be substituted, e.g., integer variable or statement label.

3. Square brackets ([ ]) enclose optional items.

4. An ellipsis (...) indicates that the preceding item or bracketed group can be repeated any number of times.

For example, if the description were

```
CALL sub [ (a[,a]...) ]
```

then all of the following would be correct:

```
CALL TIMER
CALL INSPCT (I,J,3.0)
CALL REGRES (A)
```

This page intentionally left blank.

CHAPTER 1

INTRODUCTION TO PDP-11 FORTRAN

1.1 LANGUAGE OVERVIEW

The PDP-11 FORTRAN language conforms to the specifications for American National Standard FORTRAN X3.9-1966. The following enhancements to American National Standard FORTRAN are included in PDP-11 FORTRAN:

1. Any arithmetic expression can be used as an array subscript. If the expression is not of type Integer, it is converted to Integer form.

2. Alphanumeric literals (character strings bounded by apostrophes) can be used in place of Hollerith constants.

3. Mixed-mode expressions can contain elements of any data type, including Complex.

4. The statement label list in an assigned GO TO statement is optional.

5. The following Input/Output statements have been added:

ACCEPT TYPE PRINT	Device-oriented I/O
DEFINE FILE READ (u'r) WRITE (u'r) FIND (u'r)	Unformatted Direct Access I/O

6. The specifications END=n and/or ERR=n can be included in any READ or WRITE statement to transfer control to the specified statement upon occurrence of an end-of-file or error condition.

7. The following additional data type is provided:

   LOGICAL*1

8. The IMPLICIT statement redefines the implied data type of symbolic names.

9. Any FORTRAN statement can be followed, in the same line, by an explanatory comment that begins with an exclamation point.

10. Statements can be included in a program for debugging purposes by placing the letter D in column 1. These statements are compiled only when the associated compiler option switch is set; otherwise, they are treated as comments.

11. Undersized input data fields can contain external field separators to override the FORMAT field width specifications for those fields (called "short field termination").

12. Any arithmetic expression can be used as the initial value, increment, or limit parameter in the DO statement, or as the control parameter in the computed GO TO statement.

13. The value of the DO statement increment parameter can be negative.

14. Constants and expressions are permitted in the I/O lists of WRITE, TYPE, and PRINT statements.

15. A PROGRAM statement can be used in a main program.

16. An optional comma is allowed in DO statements for better readability and reduced user errors.

    For example

```
DO 5, I=1,10
```

In addition, FORTRAN IV-PLUS includes the following extensions:

17. PARAMETER statements can be used to give symbolic names to constants.

18. ENTRY statements can be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single program unit.

19. Lower bounds for array dimensions can be specified in all array declarators.

20. The data type INTEGER*4 provides a sign plus 31 bits of precision.

21. The following Input/Output statements have been added:

```
OPEN                 }
CLOSE                } File control and attribute specification
READ  (u'r,fmt)      }
WRITE (u'r,fmt)      } Formatted Direct Access I/O
```

22. Generic function selection by argument data type is provided for many FORTRAN supplied functions.

23. The control variable of a DO statement can be a REAL or DOUBLE PRECISION variable.

24. The INCLUDE statement incorporates FORTRAN source text from a separate file into a FORTRAN program.

25. Formatted input/output can be performed without a format specification using list-directed I/O.

1.2 ELEMENTS OF A FORTRAN PROGRAM

A FORTRAN program consists of FORTRAN statements and optional comments. The statements are organized in logical units called program units. A program unit is a sequence of statements terminated by an END statement that defines a computing procedure. A program unit can be either a main program or a subprogram. One main program and possibly one or more subprograms comprise the executable program.

1.2.1 Statements

Statements are grouped into two general classes: executable and nonexecutable. Executable statements describe the action of the program; nonexecutable statements describe data arrangement and characteristics, and provide editing and data conversion information.

Statements are divided into physical sections called lines. A line is a string of up to 72 characters. If a statement is too long to be contained on one line, it can be continued on one or more additional lines, called continuation lines. A continuation line is identified by the presence of a continuation character in the sixth column of that line. (For further information concerning continuation characters, see Section 1.3.4, Continuation Field.)

A statement can be identified by a statement label so that other statements can refer to it, either for the information it contains or to transfer control to it. A statement label has the form of an integer number placed in the first five columns of a statement's initial line.

1.2.2 Comments

Comments do not affect the meaning of the program in any way, but are a documentation aid to the programmer. They should be used freely to describe the actions of the program, to identify program sections and processes, and to provide greater ease in reading the source program listing. The letter C in the first column of a source line identifies that line as a comment. Also, if an exclamation point (!) is placed in the statement portion of a source line, the rest of that line is treated as a comment.

1.2.3 The FORTRAN Character Set

The FORTRAN character set consists of:

1. The upper case letters A through Z and the lower case letters a through z

2. The numerals 0 through 9

3. The following special characters:

Character	Name
Δ	Space or Blank or Tab
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe
"	Double Quote
$	Dollar Sign
:	Colon

In FORTRAN IV-PLUS, the following additional special characters are used:

<	Left Angle Bracket
>	Right Angle Bracket

Other printable characters may appear in a FORTRAN statement only as part of a Hollerith constant or alphanumeric literal. Any printable character can appear in a comment.

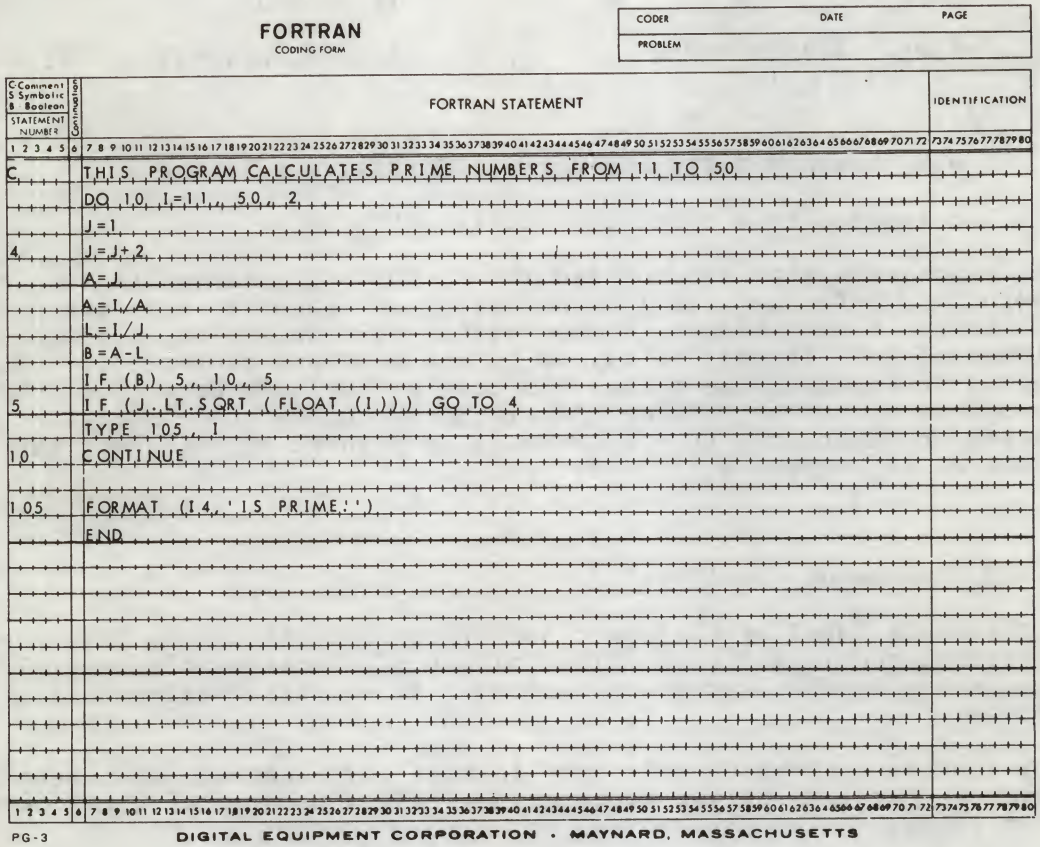
Except in alphanumeric literals and Hollerith constants, the compiler makes no distinction between upper and lower case letters.

1.3 FORMATTING A FORTRAN LINE

The formatting of a FORTRAN line is the same for lines punched into cards or paper tape and for lines entered from a terminal using a text editor. Only the method of formatting differs.

1.3.1 Using FORTRAN Coding Forms

A FORTRAN line is divided into fields for statement labels, continuation indicators, statement text and sequence numbers. Each column represents a single character. The usage of each type of field is described in the following sections.



FORTRAN
CODING FORM

CODER | DATE | PAGE
PROBLEM

C Comment S Symbolic B Boolean STATEMENT NUMBER	Continuation	FORTRAN STATEMENT	IDENTIFICATION
1 2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80
C		THIS PROGRAM CALCULATES PRIME NUMBERS FROM 1 TO 50	
		DO 10 I=1,,50,,2	
		J=1	
4		J=J+2	
		A=J	
		A=I/A	
		L=I/J	
		B=A-L	
		IF (B) 5,,10,,5	
5		IF (J.LT.SQRT (FLOAT (I))) GO TO 4	
		TYPE 105,, I	
10		CONTINUE	
105		FORMAT (I4,' IS PRIME')	
		END	

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

PG-3 DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS

Figure 1-1
FORTRAN Coding Form

1.3.2 Using a Text Editor

When creating a source program via a terminal, using a text editor, the user can type the lines on a "character-per-column" basis, or use the TAB character to facilitate formatting the lines.

If a TAB character appears at the beginning of a line, possibly preceded by a statement label or a D in column 1, and the following character is a digit, the compiler treats the digit as a continuation indicator and the character following as the beginning of the statement text. If the character following the initial TAB is not a digit, the compiler treats it as the first character of the statement text. If the continuation indicator is a "0" the line is an initial line.

While many text editors and terminals advance the terminal print carriage to a predefined print position when the TAB character is typed, this action is not related to the interpretation of the TAB character described above.

Formatting of the following lines can be accomplished in either of the following ways:

→|1ΔHOLD,MOVE,DECODE	or	ΔΔΔΔΔ1ΔHOLD,MOVE,DECODE
C→|INITIALIZEΔARRAYS	or	CΔΔΔΔΔINITIALIZEΔARRAYS
10→|W=3	or	10ΔΔΔΔW=3
→|SEL(1)=111200022D0	or	ΔΔΔΔΔΔSEL(1)=111200022D0

where:

→| represents a TAB character (CTRL/I), and

Δ represents a space character (SPACE bar).

The space character can be used in a FORTRAN statement to improve legibility of the line; the compiler ignores all spaces in a statement field except those within a Hollerith constant or alphanumeric literal (e.g., GO TO and GOTO are equivalent). The TAB character in a statement field is treated the same as a space by the compiler. In the source listing produced by the compiler, the TAB causes the character that follows to be printed at the next tab stop (located at columns 9,17,25,33, etc.).

1.3.3 Statement Label Field

A statement label or statement number consists of one to five decimal digits placed in the first five columns of a statement's initial line. Spaces and leading zeros are ignored. An all-zero statement label is illegal.

Any statement to which reference is made by another statement must have a label. No two statements within a program unit can have the same label.

1.3.3.1 Comment Indicator - The letter C can be placed in column one to indicate that the line is a comment. The compiler prints the contents of that line in the source program listing, then ignores the line.

1.3.3.2 Debug Statement Indicator - Debug statements are designated by a D in column one. The initial line of the debug statement can contain a statement label in columns two through five. If a debug statement is continued onto more than one line, then every continuation line must contain a D in column one as well as a continuation character in column six.

The debug statement can be treated either as source text to be compiled or as a comment, depending on the setting of a compiler command switch. When the switch is set, debug statements are compiled as a part of the source program; when it is not set, debug statements are treated as comments.

1.3.4 Continuation Field

Column six of a FORTRAN line is reserved for a continuation indicator. Any character except zero or space in this column is recognized as a continuation indicator. A statement can be divided into distinct lines at any point. The characters beginning in column seven of a continuation line are considered to follow the last character of the previous line as if there were no break at that point.

Comment lines cannot be continued. All comment lines must begin with the letter C in column one. Comment lines must not intervene between a statement's initial line and its continuation line(s), or between successive continuation lines.

1.3.5 Statement Field

The text of a FORTRAN statement is placed in columns 7 through 72. Because the compiler ignores the TAB character and spaces (except in Hollerith constants and alphanumeric literals), the user can space the text in any way desired for maximum legibility.

1.3.6 Sequence Number Field

A sequence number or other identifying information can appear in columns 73-80 of any line in a FORTRAN program. The characters in this field are ignored by the compiler.

NOTE

Text is ignored with no warning message printed if a line accidentally extends beyond character position 72.

1.4 PROGRAM UNIT STRUCTURE

Figure 1-2 provides a graphic representation of the rules for statement ordering. In this figure, vertical lines separate statement types which may be interspersed, such as DATA and executable statements; horizontal lines indicate statement types that can not be interspersed, such as DATA and PARAMETER statements.

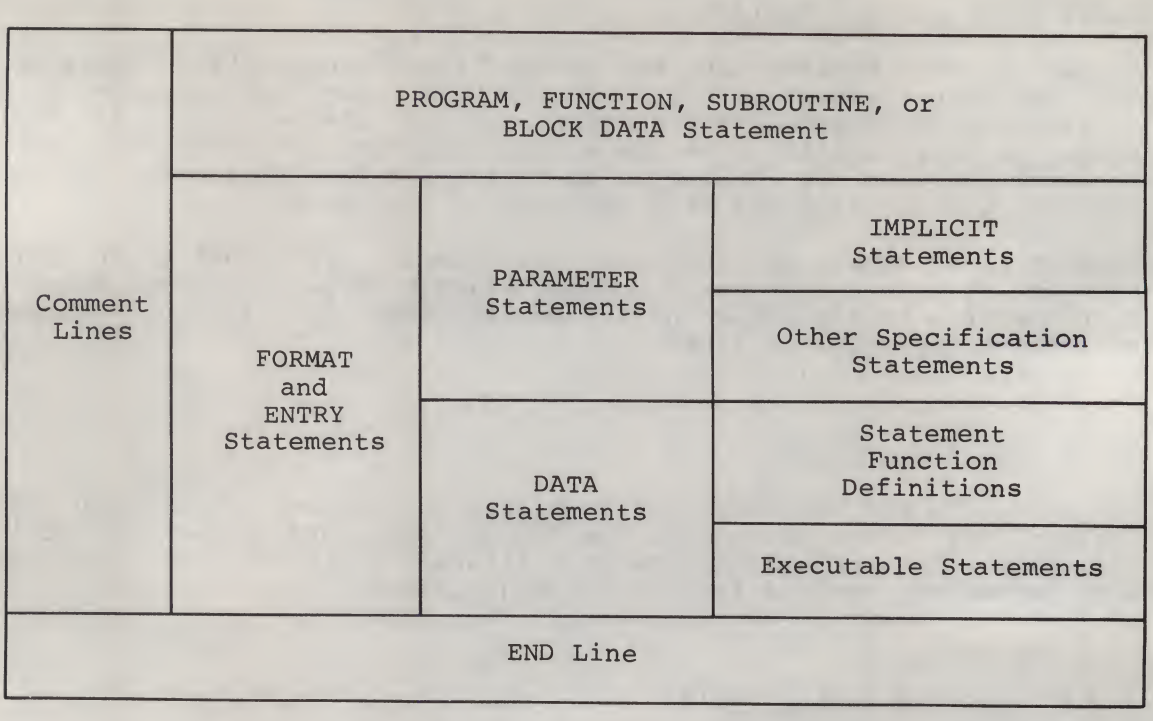


Figure 1-2
Required Order of Statements and Lines

INCLUDE

1.5 INCLUDE STATEMENT

The INCLUDE statement specifies that the contents of a designated file are to be incorporated in the FORTRAN compilation directly following the INCLUDE statement.

The INCLUDE statement has the form:

```
INCLUDE 'filspec[/[NO]LIST]'
```

filspec is a character string that specifies the file to be included in the compilation. This file specification must be acceptable to the operating system. (See the FORTRAN IV-PLUS User's Guide for the form of a file specification.)

The option specification /LIST indicates that the statements in the included file are to be listed in the compilation source listing preceded by an asterisk (*). The /NOLIST option specification indicates that the included statements are not to be listed in the compilation source listing. The /LIST option is assumed if neither option is specified.

When the compiler encounters an INCLUDE statement, it stops reading statements from the current file and compiles the statements in the included file. When the end of the included file is reached, compilation resumes with the statement following the INCLUDE statement.

An INCLUDE statement can be contained in an included file.

An included file can not begin with a continuation line. Each FORTRAN statement must be completely contained within a single file.

The INCLUDE statement can appear anywhere that a comment line can appear as illustrated in Figure 1-2. Any FORTRAN statement can appear in an included file; however, the included statements, when combined with the other statements of the compilation must satisfy the statement ordering restrictions of Section 1.4.

Example

```
          Main Program File                    File COMMON.FTN

          INCLUDE 'COMMON.FTN'                 PARAMETER M = 100
          DIMENSION Z(M)                       COMMON X(M),Y(M)
          CALL CUBE
          DO 5, I=1,M
     5    Z(I) = X(I)+SQRT(Y(I))
               .
               .
               .

          SUBROUTINE CUBE
          INCLUDE 'COMMON.FTN'
          DO 10, I=1,M
    10    X(I) = Y(I)**3
          RETURN
          END
```

THE FILE COMMON.FTN defines the size of the blank COMMON block and the size of the arrays X, Y and Z.

CHAPTER 2

FORTRAN STATEMENT COMPONENTS

2.1 INTRODUCTION TO FORTRAN STATEMENT COMPONENTS

The basic components of FORTRAN statements are:

1. Constants
   A constant represents a fixed, self-describing value.

2. Variables
   A variable is a symbolic name that represents a stored value.

3. Arrays
   An array is a group of values, stored contiguously, that can be referred to individually or collectively. The individual values are called array elements. A symbolic name is used to refer to the array.

4. Expressions
   An expression can be a single constant, variable, array element, or function reference, or it can be a combination of those components and certain other elements, called operators, that specify computations to be performed on the values represented by those components to obtain a single result.

5. Function References
   A function reference is the name of a function followed by a list of arguments which performs the computation indicated by the function definition. The resulting value is used in place of the function reference. Function references are treated in detail in Chapter 8.

2.2 SYMBOLIC NAMES

Symbolic names are used to identify many entities within a FORTRAN program unit.

A symbolic name is a string of letters and digits, the first of which must be a letter. The name can be of any length, but characters after the sixth are ignored. Examples of valid and invalid symbolic names are:

Valid	Invalid	
NUMBER	5Q	(Begins with a numeral)
K9	B.4	(Contains a special character)
X		

Table 2-1 indicates the types of entities which are identified by symbolic names.

Within one program unit, the same symbolic name cannot be used to identify more than one entity, except as noted. Within an executable program, the same symbolic name can be used to identify only one of the entities indicated as "Unique in Executable Program" in Table 2-1.

Each entity indicated as "Typed" in Table 2-1 has a data type. The means of specifying the data type of a name are discussed in Sections 2.5.1 and 2.5.2.

Within a subprogram, symbolic names are also used as dummy arguments. A dummy argument can represent a variable, array, array element, constant, expression, or subprogram.

Table 2-1
Classes of Symbolic Names

Entity	Typed	Unique in Executable Program
Variables	yes	no
Arrays	yes	no
Arithmetic statement functions	yes	no
Processor-defined functions	yes	yes
Function subprograms	yes	yes
Subroutine subprograms	no	yes
Common blocks	no	yes
Main programs	no	yes
Block data subprograms	no	yes
Function entries	yes	yes
Subroutine entries	no	yes
Parameter constants	yes	no

2.3 DATA TYPES

Each basic component represents data of one of several different types. The data type of a component can be inherent in its construction, implied by convention, or explicitly declared. The data types available in FORTRAN, and their definitions, are as follows:

1. Integer - A whole number.
2. Real - A decimal number; it can be a whole number, a decimal fraction, or a combination of the two.
3. Double Precision - Similar to real, but with more than twice the degree of accuracy in its representation.
4. Complex - A pair of real values that represents a complex number; the first represents the real part of that number, the second represents the imaginary part.
5. Logical - The logical value "true" or "false".

An important attribute of each type of data is the amount of memory required to represent a value of that type. Variations on the basic types affect either the accuracy of the represented value or the allowed range of values.

Hollerith constants and alphanumeric literals have no data type. They assume the data type of the context in which they appear. See Section 2.4.7.2 for details.

Standard FORTRAN specifies that a "storage unit" is the amount of storage needed to represent a Real, Integer or Logical value. Double Precision and Complex values occupy two storage units. In PDP-11 FORTRAN a storage unit corresponds to four bytes (two words) of memory.

PDP-11 FORTRAN provides additional types of data for optimum selection of performance and memory requirements. Table 2-2 lists the data types available, the names associated with each data type, and the amount of storage required. The form *n appended to a data type name is called a data type length specifier.

Table 2-2
Data Type Storage Requirements

DATA TYPE	FORTRAN IV (Bytes)	FORTRAN IV-PLUS (Bytes)
INTEGER	2 or 4 (Note 1)	2 or 4 (Note 2)
INTEGER*2	2	2
INTEGER*4	4 (Note 3)	4
REAL REAL*4	4	4
DOUBLE PRECISION REAL*8	8	8
COMPLEX COMPLEX*8	8	8
LOGICAL	4	2 or 4 (Note 4)
LOGICAL*1 BYTE	1 (Note 5)	1 (Note 5)
LOGICAL*2	Not Available	2
LOGICAL*4	4	4

NOTES:

1. Either two or four bytes are allocated according to a compiler command specification. The default allocation is two bytes. In either case only two bytes are used to represent the integer value. The 4-byte allocation is provided to simplify use of programs developed on other FORTRAN systems. (Consult the FORTRAN IV User's Guide for details.)

2. Either two or four bytes are allocated depending on a compiler command specification. The default allocation is two bytes. When four bytes are allocated, all four bytes are used to represent the integer value. Hence 4-byte integers can store a larger range of values than 2-byte integers. (4-byte integers can also simplify the use of programs developed on other FORTRAN systems.)

3. Four bytes are allocated but only the first two are used to represent the integer value. The range of possible values is therefore the same as for INTEGER*2 variables.

4. Either two or four bytes are allocated depending on a compiler command specification. Two bytes is the normal case.

5. The 1-byte storage area can contain the logical values true or false, a single Hollerith character, or integers in the range -128 to +127.

Additional descriptions of these data types and their representations are presented in the sections that follow.

2.4 CONSTANTS

A constant represents a fixed value. A constant can represent a numeric value, a logical value, or a character string.

2.4.1 Integer Constants

An integer constant is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

An integer constant has the form:

snn

where nn is a string of numeric characters and s is an optional sign. Leading zeros, if any, are ignored.

A negative integer constant must be preceded by a minus symbol; a positive constant can optionally be preceded by a plus symbol (an unsigned constant is presumed to be positive).

Except for a leading algebraic sign, an integer constant cannot contain any character other than the numerals 0 through 9.

The absolute value of an integer constant cannot be greater than 32767 in FORTRAN IV or 2147483647 in FORTRAN IV-PLUS.

Examples

Valid Integer Constants	Invalid Integer Constants	
0	99999999999	(Too large)
-127	3.14	(Decimal point and
+32123	32,767	comma not allowed)

In FORTRAN IV-PLUS, if the value of the constant is within the range -32768 to +32767 then it represents a 16-bit signed quantity and is treated as INTEGER*2 data type. If the value is outside that range then it represents a 32-bit signed quantity and is treated as INTEGER*4 data type.

2.4.2 Real Constants

A basic real constant is a string of decimal digits with a decimal point.

A basic real constant has the form:

s.nn OR snn.nn OR snn.

where nn is a string of numeric characters and s is an optional sign. The decimal point can appear anywhere in the string. The number of digits is not limited, but only the leftmost eight digits are significant. Leading zeros (zeros to the left of the first non-zero digit) are ignored when counting the leftmost eight digits. Thus in the constant 0.000012345678 all of the non-zero digits are significant.

A basic real constant must contain a decimal point.

A real constant is a basic real constant or an integer constant or basic real constant followed by a decimal exponent of the form:

Esnn

where nn is a 1- or 2-digit integer constant and s is an optional sign. It represents a power of ten by which the preceding real or integer constant is to be multiplied (e.g., 1E6 represents the value 1.0×10^6).

A real constant occupies two words (i.e., four bytes) of PDP-11 storage and is interpreted as a real number having a degree of precision slightly greater than seven decimal digits.

A minus symbol must appear between the letter E and a negative exponent; a plus symbol is optional for a positive exponent.

Except for algebraic signs, a decimal point, and the letter E (if used), a real constant cannot contain any character other than the numerals 0 through 9.

If the letter E appears in a real constant, a 1- or 2-digit integer constant must follow; the exponent field cannot be omitted, but can be zero.

The magnitude of a real constant cannot be smaller than 0.29×10^{-38} nor greater than 1.7×10^{38}.

Examples

Valid Real Constants	Invalid Real Constants	
3.14159	1,234,567	(Commas not allowed)
621712.	325E-75	(Too small)
-.00127	-47.E47	(Too large)
+5.0E3	100	(Decimal point missing)
2E-3	$25.00	(Special character not allowed)

2.4.3 Double Precision Constants

A double precision constant is a basic real constant or an integer constant followed by a decimal exponent of the form:

Dsnn

where nn is a 1- or 2-digit integer constant and s is an optional sign. The number of digits that precede the exponent is not limited, but only the leftmost 17 digits are significant.

A double precision constant occupies four words of PDP-11 storage and is interpreted as a real number having a degree of precision approximately equal to 17 significant digits.

A negative double precision constant must be preceded by a minus symbol; a positive constant can optionally be preceded by a plus symbol. Similarly, a minus symbol must appear between the letter D and a negative exponent; a plus symbol is optional for a positive exponent.

The exponent field following the letter D cannot be omitted, but can be zero.

The magnitude of a double precision constant cannot be smaller than 0.29×10^{-38}, nor greater than 1.7×10^{38}.

Examples

```
1234567890D+5
+2.71828182846182D00
-72.5D-15
  1D0
```

2.4.4 Complex Constants

A complex constant is a pair of real constants separated by a comma and enclosed in parentheses. The first real constant represents the real part of that number and the second represents the imaginary part.

A complex constant has the form:

```
(rc,rc)
```

where rc is a real constant. The parentheses and comma are part of the constant and must be present. The rules for the constituent real constants are given in Section 2.4.2.

A complex constant occupies four consecutive words of storage and is interpreted as a complex number.

Examples

```
(1.70391,-1.70391)
(+12739E3,0.)
```

2.4.5 Octal Constants

An octal constant is an alternate way of representing an integer constant and can be used in a like manner.

An octal constant has the form:

```
"nn
```

where nn is a string of octal digits.

Except for the leading double quote, which must be present, an octal constant cannot contain any character other than the numerals 0 through 7.

An octal constant cannot be smaller than zero, nor greater than 177777 in FORTRAN IV or 37777777777 in FORTRAN IV-PLUS.

Examples

Valid Octal Constants	Invalid Octal Constants	
"7213	32767	(Double quote missing)
"1	"184	(Illegal character)
"17776		

2.4.6 Logical Constants

A logical constant specifies a logical value, "true" or "false". Therefore, there are only two possible logical constants. They appear as:

.TRUE.

and

.FALSE.

The delimiting periods are part of each constant and must be present.

2.4.7 Hollerith Constants

A Hollerith constant is a string of ASCII characters preceded by a character count and the letter H.

A Hollerith constant has the form:

$nHc_1c_2c_3 \ldots c_n$

where n is an unsigned, non-zero integer constant stating the number of characters in the string (including spaces and tabs), and each c is an ASCII character. The maximum number of characters is 255.

Hollerith constants are stored as byte strings, one character per byte.

Hollerith constants have no data type. They assume the data type of the context in which they appear.

Examples

Valid Hollerith Constants	Invalid Hollerith Constant
16HTODAY'SΔDATEΔIS: 1HΔ	3HABCD (Wrong number of characters)

2.4.7.1 Alphanumeric Literals - An alphanumeric literal is an alternate form of Hollerith constant.

An alphanumeric literal has the form:

$'c_1c_2c_3 \ldots c_n'$

where each c is a printable ASCII character. Both delimiting apostrophes must be present. The maximum number of characters in an alphanumeric literal is 255.

Within an alphanumeric literal, the apostrophe character is represented by two consecutive apostrophes.

Examples

'CHANGEΔPRINTERΔPAPERΔTOΔPREPRINTEDΔFORMΔNO.Δ721'

'TODAY''SΔDATEΔIS:ΔΔ'

2.4.7.2 Data Type Rules for Hollerith Constants - When an alphanumeric literal or Hollerith constant is used in an expression, the data type assumed for the constant is governed by the following rules.

1. In combination with a binary operator, including the assignment operator, the type of the constant is the type of the other operand.

   Examples:

Statement	Data Type	Length of Constant
REL = 'ABCD'	REAL*4	4
IF(I.EQ.'XY') GO TO 3	INTEGER*2	2
M = N - 'ABC'	INTEGER*2	2
X = 'Z'	REAL*4	4

2. In contexts where a specified data type is required, generally integer, that type is assumed for the constant.

   Example:

Y(IX)=Y('ABC')+3.	INTEGER*2	2

3. When the constant is used as an actual argument, no data type is assumed.

   Example:

CALL APAC ('ABCDEFGHI')	9

4. In all other contexts, INTEGER*2 type is assumed.

   Examples:

IF ('AB') 1,2,3	INTEGER*2	2
I= 'C'-'A'	INTEGER*2	2
J= .NOT. 'B'	INTEGER*2	2

When the length of the constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right.

The number of characters required for each data type is illustrated in Table 2-2. Each character occupies one byte of storage.

2.4.8 Radix-50 Constants

Radix-50 is a special character data representation in which up to three characters from the Radix-50 character set (a subset of the ASCII character set) can be encoded and packed into a single PDP-11 word. Radix-50 constants can only be used in DATA statements.

A Radix-50 constant has the following form:

$$nRc_1c_2 \cdots c_n$$

where n is an unsigned non-zero integer constant that states the number of characters to follow, and each c is a character from the Radix-50 character set. The maximum number of characters is twelve. The character count must include any spaces that appear in the character string (the space character is a valid Radix-50 character).

The internal numeric value of any combination of one, two, or three Radix-50 characters is tabulated in Appendix A.

The Radix-50 characters and their code values are:

Character	Radix-50 Value (Octal)
Space	0
A - Z	1-32
$	33
.	34
(not used)	35
0 - 9	36-47

Examples

4RABCD

6RΔΔTOΔΔ

4RDK0: (Invalid; colon is not a Radix-50 character)

2.5 VARIABLES

A variable is a symbolic name that is associated with a storage location. The value of the variable is the value currently stored in that location; that value can be changed by assigning a new value to the variable. (The form of a symbolic name is given in Section 2.2.)

Variables are classified by data type, just as constants are. The data type of a variable indicates the type of data it represents, its precision, and its storage requirements. When data of any type is assigned to a variable, it is converted, if necessary, to the data type of the variable. The data type of a variable may be established either by type declaration statements, IMPLICIT statements, or predefined typing rules.

Two or more variables are associated with each other when each is associated with the same storage location; or, partially associated, when part (but not all) of the storage associated with one variable is the same as part or all of the storage associated with another

variable. Association and partial association occur through the use of COMMON statements, EQUIVALENCE statements, and through the use of actual arguments and dummy arguments in subprogram references.

A variable is said to be defined if the storage with which it is associated contains a datum of the same type as the name. A variable can be defined prior to program execution by means of a DATA statement or during execution by means of assignment or input statements.

If variables of differing types are associated (or partially associated) with the same storage location, then defining the value of one variable (for example, by assignment) causes the value of the other variable to become not defined.

2.5.1 Data Type Specification

Type declaration statements (Section 7.2) specify that given variables are to represent specified data types. For example:

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

These statements indicate that the variable VAR1 is to be associated with a 4-word storage location that is to contain complex data, and that the variable VAR2 is to be associated with a 4-word double precision storage location.

The IMPLICIT statement (Section 7.1) has a broader scope: it states that any variable having a name that begins with a specified letter, or any letter within a specified range, is to represent a specified data type, in the absence of an explicit type declaration.

The data type of a variable can be explicitly specified only once. An explicit type specification takes precedence over the type implied by an IMPLICIT statement.

2.5.2 Data Type by Implication

In the absence of any IMPLICIT statements, all variables having names beginning with I, J, K, L, M, or N are assumed to represent integer data. Variables having names beginning with any other letter are assumed to be real variables. For example:

Real Variables	Integer Variables
ALPHA	KOUNT
BETA	ITEM
TOTAL	NTOTAL

2.6 ARRAYS

An array is a group of contiguous storage locations associated with a single symbolic name, the array name. The individual storage locations, called array elements, are referenced by a subscript appended to the array name. Subscripts are discussed in Section 2.6.2.

An array can have from one to seven dimensions. A column of figures is an example of a 1-dimensional array. Several columns of figures would represent a 2-dimensional array; to refer to a specific value in this array, both its entry (or row) number and its column number must be specified. If this table of figures covered several pages, the array would be 3-dimensional. To locate a value in a 3-dimensional array, the row number, a column number, and a page (or level) number must be specified.

The following FORTRAN statements establish arrays:

1. Type declaration statements (Section 7.2),
2. DIMENSION statement (Section 7.3),
3. COMMON statement (Section 7.4), and
4. VIRTUAL statement (Appendix D)

These statements, containing array declarators (array declarators are discussed in the following sub-section), define the name of the array, the number of dimensions in the array, and the number of elements in each dimension.

2.6.1 Array Declarators

An array declarator specifies the symbolic name that identifies an array within a program unit and indicates the properties of that array.

An array declarator has the form:

```
a (d[,d] ...)
```

a — is the symbolic name of the array -- the array name. (The form of a symbolic name is given in Section 2.2.)

d — is a dimension declarator.

The number of dimension declarators indicates the number of dimensions in the array. The minimum number of dimensions is one and the maximum number is seven.

The value of a dimension declarator specifies the number of elements in that dimension. For example, a dimension declarator value of 50 indicates that the dimension contains 50 elements. The dimension declarators can be constant or variable. Variable dimension declarators are used to define adjustable arrays (see Section 2.6.6). The number of elements in an array is equal to the product of the number of elements in each dimension.

An array name can appear in only one array declarator within a program unit.

FORTRAN IV-PLUS permits the dimension declarator to specify a lower bound as well as an upper bound for any dimension of an array. This upper and lower bound dimension declarator has the form:

```
[dl:]du
```

dl is the lower bound of the dimension.

du is the upper bound of the dimension.

The value of the lower bound dimension declarator can be negative, zero, or positive. The value of the upper bound dimension declarator must not be less than the corresponding lower bound dimension declarator. The number of elements in the dimension is du-dl+1.

If the lower bound dimension declarator is omitted, it is assumed to be one.

2.6.2 Subscripts

A subscript qualifies an array name. A subscript is a list of subscript expressions enclosed in parentheses that determines which element in the array is referenced. The subscript is appended to the array name it qualifies.

A subscript has the form:

(s[,s]...)

s is a subscript expression.

In any subscripted array reference, there must be one subscript expression for each dimension defined for that array (one for each dimension declarator). For example, the following entry could be used to refer to the element located in the first row, third column, second level of the array COS in Figure 2-1 (which is the element occupying memory position 16).

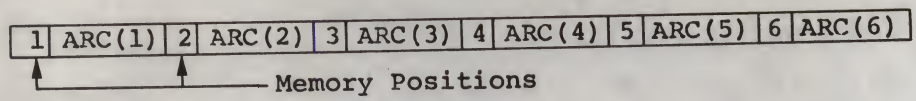
```
COS(1,3,2)
```

Each subscript expression can be any valid arithmetic expression. If the value of a subscript expression is not of type Integer, it is converted to Integer before use.

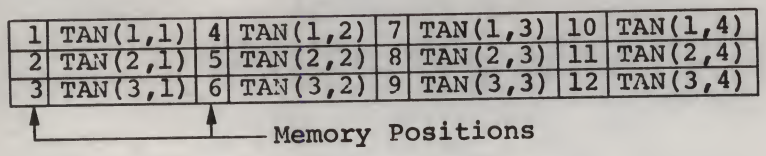
2.6.3 Array Storage

As discussed earlier in this section, it is convenient to think of the dimensions of an array as rows, columns, and levels or planes. However, the FORTRAN system always stores arrays in memory as a linear sequence of values. A 1-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multi-dimensional array is stored such that the leftmost subscripts vary most rapidly. This is called the "order of subscript progression". For example, consider the following array declarators and the arrays that they create:

1-Dimensional Array ARC(6)



2-Dimensional Array TAN(3,4)



3-Dimensional Array COS(3,3,3)

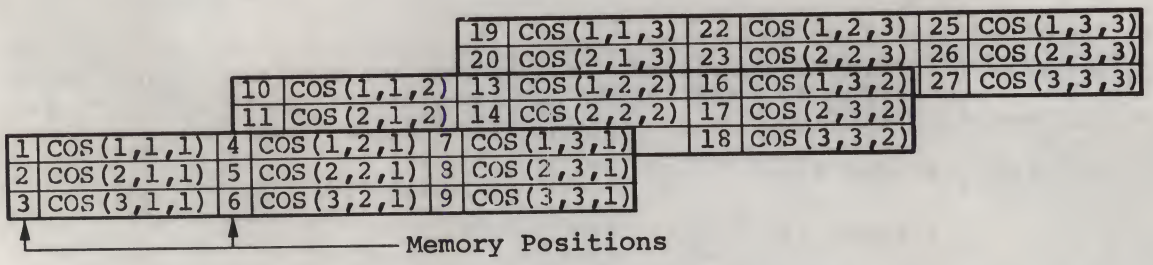


Figure 2-1
Array Storage

2.6.4 Data Type of an Array

The data type of an array is specified in the same way as the data type of a variable; that is, implicitly by the initial letter of the name, or explicitly by a type declaration statement.

All of the values in an array are of the same data type. Any value assigned to an array element is converted to the data type of the array. If an array is named in a DOUBLE PRECISION statement, for example, the compiler allocates a 4-word storage location for each element of the array. When a value of any type is assigned to any element of that array, it is converted to double precision.

The compiler stores LOGICAL*1 array elements in adjacent bytes.

2.6.5 Array References without Subscripts

In the following types of statements, an array name can appear without a subscript to specify that the entire array is to be used (or defined).

Type declaration statements

COMMON statement

DATA statement

EQUIVALENCE statement

FUNCTION statement

SUBROUTINE statement

CALL statement

Input/Output statements

Unsubscripted array names can also be used as actual arguments. The use of unsubscripted array names in all other types of statements is illegal.

2.6.6 Adjustable Arrays

Adjustable arrays are used within a single subprogram to process arrays with different dimension bounds by specifying the bounds as well as the array name as subprogram arguments.

An adjustable array declarator has variable dimension declarators. In such an array declarator, each dimension declarator must be either an integer constant or an integer dummy argument, and the array name thus declared must also appear as a dummy argument. (Consequently, adjustable array declarators can not be used in main program units.)

On entry to a subprogram containing adjustable array declarators, each dummy argument used in a dimension declarator must become associated with an integer actual argument. The values of the associated actual arguments are used together with any constants appearing in the array declarators to form an effective array declarator which determines the properties of the adjustable array for that execution of the subprogram.

The values of dummy arguments used in adjustable array declarators must not be changed within the subprogram.

The effective size of the dummy array must be equal to or less then the actual size of the associated array.

The function in the following example computes the sum of the elements of a two-dimensional array. Note the use of the parameters M and N to control the iteration.

```
      FUNCTION SUM(A,M,N)
      DIMENSION A(M,N)
      SUM = 0.0
      DO 10, I = 1,M
      DO 10, J = 1,N
10    SUM = SUM + A(I,J)
      RETURN
      END
```

Following are sample calls on SUM:

```
DIMENSION A1(10,35), A2(3,56)
SUM1 = SUM(A1,10,35)
SUM2 = SUM(A2,3,56)
SUM3 = SUM(A1,10,10)
```

In addition, dimension declarators can be expressions provided:

1. Each operand is an integer constant, or an integer dummy argument, or an integer variable in a COMMON block which is defined on entry to the subprogram.

2. Each operator is one of the operators +, -, *, /, or **. (Array references and function references are not permitted.)

Note that the upper and lower bound values are determined once each time a subprogram is entered and will not change during the execution of that subprogram even if the values of variables contained in the array declaration are changed. For example, in

```
DIMENSION ARRAY (9,5)
L=9
M=5
CALL SUB(ARRAY,L,M)
END

SUBROUTINE SUB(X,I,J)
DIMENSION X(-I/2:I/2,J)
J = 1
I = 2
END
```

the adjustable array X will be declared as X(-4:4,5) on entry to subroutine SUB. The assignments to I and J will not affect that declaration.

2.7 EXPRESSIONS

An expression represents a single value. It can be a single basic component, such as a constant or variable, or a combination of basic components with one or more operators. Operators specify computations to be performed, using the values of the basic components, to obtain a single value.

Expressions can be classified as arithmetic, relational, or logical. Arithmetic expressions yield numeric values; relational and logical expressions produce logical values.

2.7.1 Arithmetic Expressions

Arithmetic expressions are formed with arithmetic elements and arithmetic operators. The evaluation of such an expression yields a single numeric value.

An arithmetic element may be any of the following:

1. A numeric constant
2. A numeric variable
3. A numeric array element
4. An arithmetic expression enclosed in parentheses
5. An arithmetic function reference (Functions and function references are described in Chapter 8.)

The term "numeric" in these cases can also be interpreted to include logical data, since data of this type is treated as integer data when used in an arithmetic context.

Arithmetic operators specify a computation to be performed using the values of arithmetic elements; they produce a numeric value as a result. The operators and their meanings are:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and Unary Plus
-	Subtraction and Unary Minus

The above are called binary operators, because each is used in conjunction with two elements. The + and - symbols can also be used as unary operators when written immediately preceding an arithmetic element to denote a positive or negative value.

Any arithmetic operator can be used in conjunction with any valid arithmetic element except for certain restrictions noted below.

A value must be assigned to a variable or array element before it can be used in an arithmetic expression.

The following table illustrates the permitted combinations of base and exponent data type for the exponentiation operator.

BASE	EXPONENT			
	Integer	Real	Double	Complex
Integer	Yes	No	No	No
Real	Yes	Yes	Yes	No
Double	Yes	Yes	Yes	No
Complex	Yes	No	No	No

In addition, a negative element can only be exponentiated by an integer element; an element having a value of zero cannot be exponentiated by another zero-value element.

In any valid exponentiation, the result is of the same data type as the base element, except in the case of a real base and a double precision exponent; the result in this case is double precision.

Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. The precedence of the operators is:

Operator	Precedence
**	First
* and /	Second
+ and -	Third

Whenever two or more operators of equal precedence (such as + and -) appear, they can be evaluated in any order chosen by the compiler so long as the actual order of evaluation is algebraically equivalent to a left to right order of evaluation. Exponentiation, however, is evaluated right to left. For example A**B**C is evaluated as A**(B**C).

2.7.1.1 Use of Parentheses - Parentheses can be used to override the normal evaluation order. An expression enclosed in parentheses is treated as a single arithmetic element. That is, it is evaluated first to obtain its value, then that value is used in the evaluation of the remainder of the larger expression of which it is a part. An example of the effect of parentheses is shown below (the numbers below the operators indicate the order in which the operations are performed).

```
4 + 3 * 2 - 6 / 2 = 7
    ↑   ↑   ↑   ↑
    2   1   4   3

(4+3) * 2 - 6 / 2 = 11
  ↑   ↑   ↑   ↑
  1   2   4   3

(4 + 3 * 2 - 6) / 2 = 2
  ↑   ↑   ↑     ↑
  2   1   3     4

((4+3) * 2 - 6) / 2 = 4
  ↑   ↑   ↑     ↑
  1   2   3     4
```

Evaluation of expressions within parentheses takes place according to the normal order of precedence.

Nonessential parentheses, such as in the expression

```
4 + (3*2) - (6/2)
```

have no effect on the evaluation of the expression.

The use of parentheses to specify the evaluation order is often important in high accuracy numerical programs where evaluation orders that are algebraically equivalent might not be computationally equivalent when carried out on a computer.

2.7.1.2 Data Type of an Arithmetic Expression - If every element in an arithmetic expression is of the same data type, the value produced by the expression is also of that type. If elements of different data types are combined in an expression, the evaluation of that expression and the data type of the resulting value are dependent on a rank associated with each data type. The rank assigned to each data type is as follows:

Data Type	Rank
Logical	1 (Low)
Integer	2
Real	3
Double Precision	4
Complex	5 (High)

The data type of the value produced by an operation on two arithmetic elements of differing type is the same as that of the highest-ranked element in the operation. The data type of an expression is the same as the data type of the result of the last operation in that expression. The way in which the data type of an expression is determined is as follows:

1. Integer operations - Integer operations are performed only on integer elements. (When used in an arithmetic context, octal constants and logical entities are treated as integers.) In integer arithmetic, any fraction that can result from division is truncated, not rounded. For example, the value of the expression

   ```
   1/3 + 1/3 + 1/3
   ```

   is zero, not one.

2. Real operations - Real operations are performed only on real elements or a combination of real and integer elements. Any integer elements present are converted to real type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. Note, however, that in the statement Y = (I/J)*X, an integer division operation is performed on I and J and a real multiplication is performed on the result and X.

3. Double Precision operations - Any real or integer element in a double precision operation is converted to double precision type by making the existing element the most significant portion of a double precision datum; the least significant portion is zero. The expression is then evaluated in double precision arithmetic.

NOTE

The conversion of a real element to double precision does not increase its accuracy. For example, the real number 0.3333333 becomes 0.3333333000000000 when converted, not 0.3333333333333333. Also note that real and double precision elements are only approximate representations of actual numbers. Values resulting from a real or double precision expression are only as accurate as the degree of precision for that data type.

4. Complex operations - In an operation that contains any complex element, integer elements are converted to real type as previously described. Double precision elements are converted to real type by the rounding of the least significant portion. The real element thus obtained is designated as the real part of a complex number; the imaginary part is zero. The expression is then evaluated using complex arithmetic and the resulting value is of type complex.

2.7.2 Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator. The value of the expression is either true or false, depending on whether or not the stated relationship exists.

A relational operator tests for a relationship between two arithmetic expressions. These operators are:

Operator	Relationship
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The delimiting periods are part of each operator and must be present.

Complex expressions can be related by the .EQ. and .NE. operators only. (If one complex expression is present, the other is converted to complex type.) Complex entities are equal if their corresponding real and imaginary parts are both equal.

In a relational expression, the arithmetic expressions are evaluated first to obtain their values. Those values are then compared to determine if the relationship stated by the operator exists. For example, the expression:

```
APPLE+PEACH .GT. PEAR+ORANGE
```

states the relationship, "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If that relationship does in fact exist, the value of the expression is true; if not, the expression is false.

All relational operators have the same precedence. Thus, if two or more relational expressions appear within an expression, the relational operators are evaluated from left to right. Arithmetic operators have a higher precedence than relational operators.

Parentheses can be used to alter the evaluation of the arithmetic expressions in a relational expression exactly as in any other arithmetic expression; but since arithmetic operators are evaluated before relational operators, it is unnecessary to enclose the entire arithmetic expression in parentheses.

When two expressions of different data types are compared by a relational expression, the value of the expression having the lower-ranked data type is converted to the higher-ranked data type before the comparison is made.

2.7.3 Logical Expressions

A logical expression may be a single logical element, or may be a combination of logical elements and logical operators. A logical expression yields a single logical value, true or false.

A logical element can be any of the following:

1. An Integer or Logical constant
2. An Integer or Logical variable
3. An Integer or Logical array element
4. A relational expression
5. A logical expression enclosed in parentheses
6. An Integer or Logical function reference (Functions and function references are described in Chapter 8.)

The logical operators are:

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction. The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR). The expression is true if, and only if, either A or B, or both, is true.
.XOR.	A .XOR. B	Logical exclusive OR. The expression is true if A is true and B is false, or vice versa, but is false if both elements have the same value.
.EQV.	A .EQV. B	Logical equivalence. The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.NOT.	.NOT. A	Logical negation. The expression is true if, and only if, A is false.

The delimiting periods of logical operators must be present.

When a logical operator is used to operate on logical elements, the resulting value is of type logical. When a logical operator is used with integer elements, the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements. The resulting value has type integer. When integer and logical values are combined with a logical operator, the logical value is first converted to an integer value, then the operation is carried out as for two integer elements. The resulting type is integer.

Evaluation of a logical expression is performed according to an order of precedence assigned to its operators. Some logical expressions can be evaluated without evaluating all sub-expressions; for example, if A is .FALSE. then the expression A .AND. (F(X,Y) .GT. 2.0) .AND. B is .FALSE.. The value of the expression can be determined by testing A without evaluating F(X,Y). With this method of evaluation, the function subprogram F is not necessarily called and side-effects resulting from the call can not occur.

A summary of all operators that may appear in a logical expression, and the order in which they are evaluated follows.

Operator	Evaluated
**	First
*,/	Second
+,-	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.	Eighth

Operators of equal rank are evaluated from left to right. An example of the sequence in which a logical expression is evaluated is as follows:

```
A*B+C*ABC .EQ. X*Y+DM*ZZ .AND. .NOT. K*B .GT. TT
```

is evaluated as:

```
(((A*B)+(C*ABC)).EQ.((X*Y)+(DM*ZZ))).AND.(.NOT.((K*B).GT.TT))
```

Parentheses may be used to alter the normal sequence of evaluation, just as in arithmetic expressions.

Two logical operators cannot appear consecutively, except where the second operator is .NOT..

CHAPTER 3

ASSIGNMENT STATEMENTS

Assignment statements define the value of a variable or array element by evaluating an expression and assigning the resulting value to the variable or array element.

There are three types of assignment statements:

1. Arithmetic assignment statement
2. Logical assignment statement
3. ASSIGN statement

3.1 ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement assigns the value of the expression on the right of the equal sign to the variable or array element on the left of the equal sign. The previous value of the variable, if any, is lost.

The arithmetic assignment statement has the form:

```
v = e
```

v is a numeric variable name or array element name.

e is an expression.

The equal sign does not mean "is equal to", as in mathematics. It means "is replaced by". Thus, the statement:

```
KOUNT = KOUNT + 1
```

means, "Replace the current value of the integer variable KOUNT with the sum of that current value and the integer constant 1".

Although the symbolic name to the left of the equal sign can be undefined, values must have been previously assigned to all symbolic references in the expression.

The expression must yield a value that conforms to the requirements of the variable or array element to which it is to be assigned (for example, a real expression that produces a value greater than 32767 is illegal if the entity on the left of the equal sign is an INTEGER*2 variable).

If the data type of the variable or array element on the left of the equal sign is the same as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the entity on the left of the equal sign before it is assigned. A summary of data conversions on assignment is shown in Table 3-1.

Table 3-1
Conversion Rules for Assignment Statements

VARIABLE OR ARRAY ELEMENT (V) \ EXPRESSION (E)	INTEGER, LOGICAL, OR OCTAL CONSTANT	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncate real part of E to integer and assign to V; imaginary part of E is not used
REAL	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS portion of E to V; LS portion of E is rounded	Assign real part of E to V; imaginary part of E is not used
DOUBLE PRECISION	Append fraction (.0) to E and assign to MS portion of V; LS portion of V is zero	Assign E to MS portion of V; LS portion of V is zero	Assign E to V	Assign real part of E to MS portion of V; LS portion of V is zero, imaginary part of E is not used
COMPLEX	Append fraction (.0) to E and assign to real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part of V is 0.0	Assign MS portion of E to real part of V; LS portion of E is rounded; imaginary part of V is 0.0	Assign E to V

MS = Most Significant (high-order)

LS = Least Significant (low-order)

Examples

Valid Statements

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))

PI = 3.14159

SUM = SUM+1.
```

Invalid Statements

```
3.14 = A-B                  (Entity on the left must be a
                             variable or array element.)

-J = I**4                   (Entity on the left must not be
                             signed.)

ALPHA = ((X+6)*B*B/(X-Y)    (Invalid; left and right
                             parentheses do not balance.)
```

3.2 LOGICAL ASSIGNMENT STATEMENT

The logical assignment statement is similar to the arithmetic assignment statement, but operates with logical data. The logical assignment statement evaluates the expression on the right side of the equal sign and assigns the resulting logical value to the variable or array element on the left.

The logical assignment statement has the form:

```
v = e
```

v is a variable or array element of type Logical.

e is a logical expression.

The variable or array element on the left of the equal sign must be of type Logical; its value can be undefined.

Values, either numeric or logical, must have been previously assigned to all symbolic references that appear in the expression. The expression must yield a logical value.

Examples

```
PAGEND = .FALSE.

PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND

ABIG = A .GT. B .AND. A .GT. C .AND. A .GT. D
```

ASSIGN

3.3 ASSIGN STATEMENT

The ASSIGN statement is used to associate a statement label with an integer variable. The variable can then be used as a transfer destination in a subsequent assigned GO TO statement (see Section 4.1.3).

The ASSIGN statement has the form:

```
ASSIGN s TO v
```

s is the label of an executable statement in the same program unit as the ASSIGN statement. (It must not be the label of a FORMAT statement.)

v is an integer variable.

The ASSIGN statement assigns the statement number to the variable in a manner similar to that of an arithmetic assignment statement, with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

The ASSIGN statement must be executed before the assigned GO TO statement(s) in which the assigned variable is to be used. The ASSIGN statement and the assigned GO TO statement(s) must occur in the same program unit.

The statement

```
ASSIGN 100 TO NUMBER
```

associates the variable NUMBER with the statement label 100. Arithmetic operations on the variable, such as in the statement

```
NUMBER = NUMBER+1
```

then become invalid, since a statement label cannot be altered. The statement:

```
NUMBER = 10
```

dissociates NUMBER from statement 100, assigns it an integer value 10, and returns it to its status as an integer variable. It can no longer be used in an assigned GO TO statement.

Examples

```
ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR    (ERROR must have been defined as an
                        integer variable.)
```

CHAPTER 4

CONTROL STATEMENTS

Statements are normally executed in the order in which they are written. However, it is frequently desirable to interrupt the normal program flow by transferring control to another section of the program or to a subprogram. Transfer of control from a given point in the program may occur every time that point is reached in the program flow, or may be based on a decision made at that point.

Transfer of control, whether within a program unit or to another program unit, is performed by control statements. These statements also govern iterative processing, suspension of program execution, and program termination. The various types of control statements are:

GOTO
IF
DO
CONTINUE
CALL
RETURN
PAUSE
STOP
END

4.1 GO TO STATEMENTS

GO TO

GO TO statements transfer control within a program unit, either to the same statement every time or to one of a set of statements, based on the value of an expression.

The three types of GO TO statements are:

1. Unconditional GO TO statement
2. Computed GO TO statement
3. Assigned GO TO statement

4.1.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same statement every time it is executed.

The unconditional GO TO statement has the form:

```
GO TO s
```

s is the label of an executable statement in the same program unit as the GO TO statement.

The unconditional GO TO statement transfers control to the statement identified by the specified label. The statement label must identify an executable statement in the same program unit as the GO TO statement.

Examples

```
GO TO 7734

GO TO 99999

GO TO 27.5      (Invalid; the statement label is improperly
                formed.)
```

4.1.2 Computed GO TO Statement

The computed GO TO statement transfers control to a statement based on the value of an expression within the statement.

The computed GO TO statement has the form:

```
GO TO (slist)[,] e
```

slist is a list of one or more executable statement labels separated by commas. The list of labels is called the transfer list.

e is an arithmetic expression the value of which falls within the range 1 to n (where n is the number of statement labels in the transfer list).

The computed GO TO statement evaluates the expression e and, if necessary, converts the resulting value to integer type. The GO TO statement then transfers control to the e'th statement label in the transfer list. That is, if the list contains (30,20,30,40), and the value of e is 2, the GO TO statement transfers control to statement 20, and so on.

If the value of the expression is less than 1, or greater than the number of labels in the transfer list, control transfers to the first executable statement following the computed GO TO.

Examples

```
GO TO (12,24,36),INCHES

GO TO (320,330,340,350,360) SITU(J,K)+1
```

4.1.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement label that is represented by a variable. Because the relationship between the variable and a specific statement label must be established by an ASSIGN statement, the transfer destination may be changed, depending upon which ASSIGN statement was most recently executed.

The assigned GO TO statement has the form:

```
GO TO v[[,](slist)]
```

v — is an integer variable.

slist — (when present) is a list of one or more executable statement labels separated by commas.

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to the variable v by an ASSIGN statement.

The variable v must be of Integer type and must have been assigned a statement label value by an ASSIGN statement (not an arithmetic assignment statement) prior to the execution of the GO TO statement.

The assigned GO TO statement and its associated ASSIGN statement(s) must exist in the same program unit. Statements to which control is transferred must be executable statements in the same program unit.

Examples

```
GO TO IGO

GO TO INDEX, (300,450,1000,25)
```

In FORTRAN IV-PLUS, if the statement label value of v is not present in the list slist (and a list is specified), control transfers to the next executable statement following the assigned GO TO statement.

IF

4.2 IF STATEMENTS

An IF statement causes a conditional control transfer or the conditional execution of a statement. There are two types of IF statements:

1. Arithmetic IF statement
2. Logical IF statement

In either type, the decision to transfer control or to execute the statement is based on the evaluation of an expression within the IF statement.

4.2.1 Arithmetic IF Statement

The arithmetic IF statement is used for conditional control transfers. It can transfer control to one of three statements, based on the value of an arithmetic expression.

The arithmetic IF statement has the form:

```
IF (e) s1, s2, s3
```

e is an arithmetic expression.

s1,s2,s3 are the labels of executable statements in the same program unit.

All three labels must be present. They need not refer to three different statements. If desired, one or two labels can refer to the statement that immediately follows the IF statement.

The arithmetic IF statement first evaluates the expression in parentheses and then transfers control to one of the three statement labels in the transfer list, as follows:

If the Value is:	Control Passes to:
Less than 0	Label s1
Equal to 0	Label s2
Greater than 0	Label s3

Examples

```
IF (THETA-CHI) 50,50,100
```

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

```
IF (NUMBER/2*2-NUMBER) 20,40,20
```

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even and to statement 20 if it is odd. In this case, the third statement label must be present although it is not used, since the expression can have only negative or zero values.

4.2.2 Logical IF Statement

A logical IF statement causes a conditional statement execution. The decision to execute the statement is based on the value of a logical expression within the statement.

The logical IF statement has the form:

```
IF (e) st
```

e is a logical expression.

st is a complete FORTRAN statement. The statement can be any executable statement except a DO statement or another logical IF statement.

The logical IF statement first evaluates the logical expression. If the value of the expression is true, the contained statement is executed. If the value of the expression is false, control transfers to the next executable statement following the logical IF without executing the contained statement.

Examples

```
IF (J .GT. 4 .OR. J .LT. 1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K)*(-1D0)

IF (ENDRUN) CALL EXIT
```

DO

4.3 DO STATEMENT

DO statements are used to specify iterative processing. The DO statement causes the statements in its range to be repeatedly executed a specified number of times.

The DO statement has the form:

```
DO  s[,]  i=e1,e2[,e3]
```

s is the label of an executable statement. The statement must physically follow in the same program unit.

i is an integer variable.

e1,e2,e3 are integer expressions.

The variable i is called the control variable of the DO and e1,e2,e3 are called the initial, terminal, and increment parameters respectively. If the increment parameter is omitted, a default increment value of 1 is used.

The terminal statement of a DO loop is identified by the label that appears in the DO statement. It must not be a GO TO statement, an arithmetic IF statement, a RETURN statement, or another DO statement. A logical IF statement is acceptable as the terminal statement, provided it does not contain any of the above statements.

The statements that follow the DO statement, up to and including the terminal statement are in the range of the DO loop.

In FORTRAN IV-PLUS, the control variable can be of type INTEGER*2, INTEGER*4, REAL, or DOUBLE PRECISION. The initial, terminal, and increment parameters can be of any type and are converted, before use, to the type of the control variable, if necessary.

The DO statement first evaluates the expressions e1, e2, e3 to determine values for the initial, terminal, and increment parameters. The value of the initial parameter is then assigned to the control variable. The executable statements in the range of the DO loop are then executed repeatedly.

If the increment parameter is positive, the value of the terminal parameter must not be less than that of the initial parameter. Conversely, if the increment parameter is negative, the value of the terminal parameter must not be greater than that of the initial parameter. The value of the increment parameter must not be zero.

The number of executions of the DO range, called the iteration count, is given by

$$\left[\frac{e2 - e1}{e3} \right] + 1$$

where [X] represents the largest integer whose magnitude does not exceed the magnitude of X and whose sign is the same as that of X.

If the iteration count is zero or negative, then the loop is executed once.

4.3.1 DO Iteration Control

For each iteration of the DO loop, following execution of the terminal statement, the DO iteration control is executed:

1. The value of the increment parameter is algebraically added to the control variable.
2. The iteration count is decremented.
3. If the iteration count value is greater than zero, control is transferred to the first executable statement following the DO statement for another iteration of the range.
4. If the iteration count is zero, execution of the DO terminates.

The execution of a DO can also be terminated by a statement within the range that transfers control outside the loop. The control variable of the DO remains defined with its current value.

When execution of a DO loop terminates, if other DO loops share this terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (Section 4.3.2). If no other DO loop shares this terminal statement, or if this DO is the outermost DO, control transfers to the first executable statement following the terminal statement.

The value of the control variable, terminal parameter, or increment parameter must not be altered within the range of the DO statement. The control variable is available for reference as a variable within the range.

In FORTRAN IV-PLUS, the terminal and increment parameters can be modified within the loop without affecting the iteration count.

The range of a DO loop can contain other DO statements, as long as those "nested" DO loops conform to certain requirements (see Section 4.3.2).

Control can be transferred out of a DO loop, but cannot be transferred into a loop from elsewhere in the program. Exceptions to this rule are described in Sections 4.3.3 and 4.3.4.

Examples

```
DO 100 K=1,50,2     (25 iterations, K=49 during final
                     iteration)

DO 350 J=50,-2,-2   (27 iterations, J=-2 during final
                     iteration)

DO 25 IVAR=1,5      (5 iterations, IVAR=5 during final
                     iteration)

DO NUMBER=5,40,4    (Invalid; statement label missing)

DO 40 M=2.10        (Invalid; decimal point instead of comma)
```

The last example illustrates a common clerical error. It is a valid arithmetic assignment statement in the FORTRAN language:

```
DO40M = 2.10
```

4.3.2 Nested DO Loops

A DO loop may contain one or more complete DO loops. The range of an inner nested DO must lie completely within the range of the next outer loop. Nested loops may share the same terminal statement.

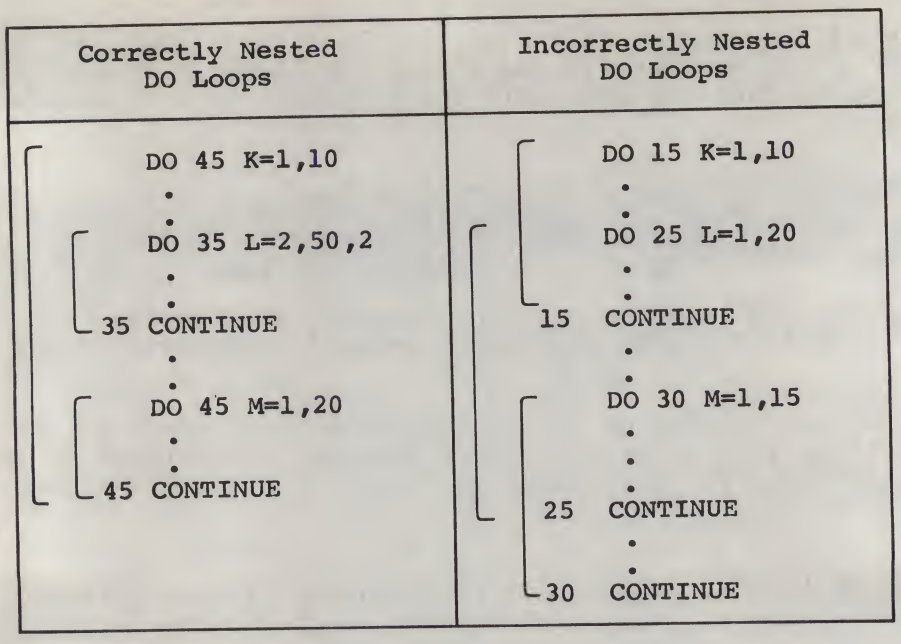


Figure 4-1
Nesting of DO Loops

4.3.3 Control Transfers in DO Loops

Within a nested DO loop structure, control can transfer from an inner loop to an outer loop. A transfer from an outer loop to an inner loop is illegal.

If two or more nested DO loops share the same terminal statement, control can be transferred to that statement only from within the range of the innermost loop. Any other transfer to that statement constitutes a transfer from an outer loop to an inner loop because the shared statement is part of the range of the innermost loop.

4.3.4 Extended Range

A DO loop is said to have an extended range if it contains a control statement that transfers control out of the loop and if, after the execution of one or more statements, another control statement returns control back into the loop. In this way the range of the loop is extended to include all of the executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

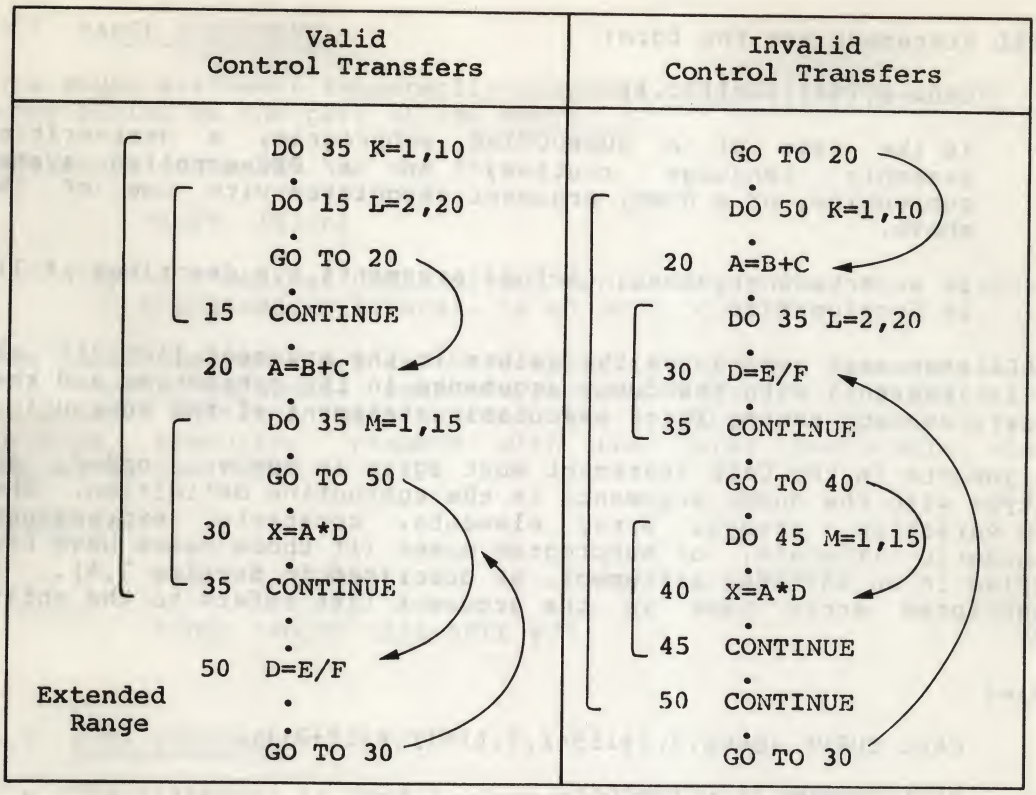


Figure 4-2
Control Transfers and Extended Range

The following rules govern the use of a DO statement extended range:

1. A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.

2. The extended range of a DO statement cannot change the control variable or parameters of the DO statement.

CONTINUE

4.4 CONTINUE STATEMENT

The CONTINUE statement simply transfers control to the next executable statement. It is used primarily as the terminal statement of a DO loop when that loop would otherwise end with a GO TO, arithmetic IF, or other prohibited control statement.

The CONTINUE statement has the form:

```
CONTINUE
```

CALL

4.5 CALL STATEMENT

The CALL statement causes the execution of a SUBROUTINE subprogram; it can also specify an argument list for use by the subroutine. (The definition and use of subroutines is treated in detail in Chapter 8.)

END

4.9 END STATEMENT

The END statement marks the end of a program unit. The END statement must be the last source line of every program unit.

The END statement has the form:

```
END
```

In a main program, if control reaches the END statement, execution of the program is terminated; in a subprogram, a RETURN statement is implicitly executed.

CHAPTER 5

INPUT/OUTPUT STATEMENTS

5.1 OVERVIEW

Input of data by a FORTRAN program is performed by READ and ACCEPT statements. Output is performed by WRITE, TYPE, and PRINT statements. Some forms of these statements are used in conjunction with format specifications which control translation and editing of the data between internal representation and character (readable) form.

Each READ or WRITE statement contains a reference to the logical unit to or from which data transfer is to take place. A logical unit can be connected to a device or file. The TYPE and ACCEPT statements have no such reference, as they cause data transfer between the processor and an implicit logical unit that is normally connected to the user's terminal. Similarly, the PRINT statement outputs data to an implicit logical unit that is normally connected to the line printer.

READ and WRITE statements fall into the following categories:

1. Unformatted Sequential I/O

   Unformatted sequential READ and WRITE statements transmit binary data without translation.

2. Formatted Sequential I/O

   Formatted sequential READ and WRITE statements transmit character data using format specifications to control the translation of data to characters on output, and to internal form on input.

3. Unformatted Direct Access I/O

   Unformatted direct access READ and WRITE statements transmit binary data without translation to and from direct access files.

4. Formatted Direct Access I/O

   Formatted direct access READ and WRITE statements transmit character data to and from direct access files using format specifications to control the translation of data to characters on output and to internal form on input.

5. List-Directed Sequential I/O

   List-directed sequential READ and WRITE statements transmit character data. The translation to internal form on input and to characters on output is controlled by the data type of the corresponding I/O list element.

Any type of READ or WRITE statement can transfer control to another statement if an error condition or end-of-file condition is detected.

The auxiliary I/O statements, REWIND, FIND, and BACKSPACE do not perform data transfer, but do file positioning functions. The ENDFILE statement writes a special form of record that will cause an end-of-file condition (and END= transfer) when read by an input statement. The DEFINE FILE statement declares a logical unit to be connected to a direct access file and specifies the characteristics of the file. Finally, there are the ENCODE and DECODE statements, which perform data transfer and translation within memory.

PDP-11 FORTRAN provides two additional auxiliary I/O statements: OPEN and CLOSE. The OPEN statement establishes a connection between a logical unit and a file or device and it declares the attributes needed for READ and WRITE operations. The CLOSE statement terminates the connection between a file or device and a logical unit.

5.1.1 Input/Output Devices and Logical Unit Numbers

PDP-11 FORTRAN uses the I/O services of the operating system and thus supports all peripheral devices that are supported by the operating system. I/O statements refer to I/O devices by means of logical unit numbers. A logical unit number is an integer constant or variable with a positive value.

Some forms of I/O statements do not contain a logical unit number. These statements use an implicit logical unit number that is system specific. Consult the appropriate FORTRAN User's Guide for details.

In FORTRAN IV-PLUS, a logical unit number can be an expression, which is converted, if necessary, to integer type prior to use.

5.1.2 Format Specifiers

Format specifiers are used in formatted I/O statements. A format specifier is either the statement label of a FORMAT statement or the name of an array containing Hollerith data interpretable as a format. Chapter 6 discusses FORMAT statements.

The asterisk character (*) can be used as a format specifier to denote list-directed formatting. Section 5.7 describes list-directed formats.

5.1.3 Input/Output Records

Input/Output statements transmit all data in terms of records. The amount of information that can be contained in one record, and the way in which records are separated, depend on the medium involved.

For unformatted I/O, the amount of data to be transmitted is specified by the I/O statement. The amount of information to be transmitted by a formatted I/O statement is determined jointly by the I/O statement and specifications in the associated format specification.

The beginning of execution of an input or output statement initiates the transmission of a new record. If an input statement requires only part of a record, the excess portion of the record is lost. In the case of formatted sequential input or output, one or more additional records can be transmitted by a single I/O statement.

5.2 INPUT/OUTPUT LISTS

An I/O list specifies the data items to be manipulated by the statement containing the list. The I/O list of an input or output statement contains the names of variables, arrays, and array elements whose values are to be transmitted. In addition, the I/O list of an output statement can contain constants and expressions.

An I/O list has the form:

```
s[,s]...
```

where each s is a simple list or an implied DO list. The I/O statement assigns input values to, or outputs values from, the list elements in the order in which they appear, from left to right.

5.2.1 Simple Lists

A simple I/O list element consists of a single variable, array reference, array element, constant, or expression. A simple I/O list consists either of a simple I/O list element or a group of two or more simple I/O list elements separated by commas and enclosed by parentheses.

When an unsubscripted array name appears in an I/O list, a READ or ACCEPT statement inputs enough data to fill every element of the array; a WRITE, TYPE, or PRINT statement outputs all of the values contained in the array. Data transmission begins with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. For example, if the unsubscripted name of a 2-dimensional array defined as:

```
ARRAY(3,3)
```

appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on, through ARRAY(3,3).

In a READ or ACCEPT statement, variables in the I/O list may be used in array subscripts later in the list. If, for example, the statement:

```
      READ (1,1250) J,K,ARRAY(J,K)
1250  FORMAT (I1,X,I1,X,F6.2)
```

was executed and the input record contained the values:

```
1,3,721.73
```

the value 721.73 is assigned to ARRAY(1,3). The first input value is assigned to J and the second to K, thereby establishing the actual subscript values for ARRAY(J,K). Variables that are to be used as subscripts in this way must appear to the left of their use in the array subscript.

Any valid expression can be included in an output statement I/O list. However, the expression must not cause further I/O operations to be attempted. A reference in an output statement I/O list expression to a function subprogram that itself performs input/output is an example of this prohibited case.

An expression must not be included in an input statement I/O list except as a subscript expression in an array reference.

5.2.2 Implied DO Lists

Implied DO lists are used to specify iteration within an I/O list, to transmit only part of an array, or to transmit array elements in a sequence other than the order of subscript progression. This type of list element functions as though it were a part of an I/O statement that resides in a DO loop, and that uses the control variable of the imaginary DO statement to specify which value or values are to be transmitted during each iteration of the loop.

An implied DO list has the form:

```
(list,i=e1,e2[,e3])
```

list	is an I/O list.
i	is a control variable definition.
e1,e2,e3	are parameter definitions.

i, e1, e2 and e3 have the same form as that used in the DO statement. The rules for the initial, terminal, and increment parameters, and for the control variable of an implied DO list are the same as those for the DO statement (see Section 4.3). An expression may be used for the initial, terminal, or increment parameter of an implied DO list, as long as it conforms to the rules in Section 4.3. The list may contain references to the control variable as long as the value of the control variable is not altered. The range of the implied DO is the list. For example:

```
      WRITE (3,200) (A,B,C, I=1,3)

      WRITE (6,15) L,M,(I,(J,P(I),Q(I,J),J=1,L),I=1,M)

      READ (1,75) (((ARRAY(M,N,I), I=2,8), N=2,8), M=2,8)
```

The first control variable definition is equivalent to the innermost DO of a set of nested loops, and therefore varies most rapidly. For example, the statement:

```
      WRITE (5,150) ((FORM(K,L), L=1,10), K=1,10,2)
150   FORMAT (F10.2)
```

is similar to:

```
      DO 50 K=1,10,2
      DO 50 L=1,10
      WRITE (5,150) FORM(K,L)
150   FORMAT (F10.2)
50    CONTINUE
```

Since the inner DO loop is executed ten times for each iteration of the outer loop, the second subscript, L, advances from one through ten for each increment of the first subscript. This is the reverse of the order of subscript progression. Also, since K is incremented by two, only the odd-numbered columns of the array will be output.

The entire list of the implied DO is transmitted before the incrementation of the control variable. For example:

```
      READ (3,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

assigns input values to the elements of arrays P and Q in the order:

```
P(1), Q(1,1), Q(1,2), ...  , Q(1,10),
P(2), Q(2,1), Q(2,2), ...  , Q(2,10),
 .      .       .             .
 .      .       .             .
P(5), Q(5,1), Q(5,2), ...  , Q(5,10)
```

When processing multidimensional arrays, it is possible to use a combination of fixed subscripts and subscripts that vary according to an implied DO. For example:

```
      READ (3,5555) (BOX(1,J), J=1,10)
```

assigns input values to BOX(1,1) through BOX(1,10), then terminates without affecting any other element of the array.

It is also possible to output the value of the control variable directly, as in the statement:

```
      WRITE (6,1111) (I, I=1,20)
```

which simply prints the integers one through twenty.

5.3 UNFORMATTED SEQUENTIAL INPUT/OUTPUT

Unformatted input and output is the transfer of data in internal (binary) format without conversion or editing. Unformatted I/O is generally used when data output by a program is to be subsequently input by the same program (or a similar program). Unformatted I/O saves execution time by eliminating the data conversion process, preserves greater precision in the external data, and usually conserves file storage space.

READ

5.3.1 Unformatted Sequential READ Statement

The unformatted sequential READ statement inputs one unformatted record from the specified logical unit and assigns the fields of the record without translation to the I/O list elements in the order in which they appear, from left to right. The amount of data each element receives is determined by its data type.

The unformatted sequential READ statement has the form:

```
READ (u[,END=s][,ERR=s])[list]
```

u is a logical unit number.

s is an executable statement label.

list is an I/O list.

An unformatted sequential READ statement reads exactly one record. If the I/O list does not use all of the values in the record, the remainder of the record is discarded. If the contents of the record are exhausted before the I/O list is satisfied, an error condition results.

If an unformatted sequential READ statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

The unformatted sequential READ statement must be used only to read records that were created by unformatted sequential WRITE statements.

Examples

```
READ (1) FIELD1, FIELD2     (Read one record from logical
                             unit 1; assign values to
                             variables FIELD1 and FIELD2.)

READ (8)                    (Advance logical unit 8 one
                             record.)
```

WRITE

5.3.2 Unformatted Sequential WRITE Statement

The unformatted sequential WRITE statement outputs the values of the elements in the I/O list to the specified logical unit without translation, as one unformatted record.

The unformatted sequential WRITE statement has the form:

```
WRITE (u[,ERR=s])[list]
```

u is a logical unit number.

s is an executable statement label.

list is an I/O list.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified unit.

Examples

```
WRITE (1) (LIST(K),K=1,5)    (Output the contents of elements
                              1 through 5 of array LIST to
                              logical unit 1.)

WRITE (4)                     (Write a null record on logical
                              unit 4.)
```

5.4 FORMATTED SEQUENTIAL INPUT/OUTPUT

Formatted input and output statements are used in conjunction with FORMAT statements (or format specifications stored in arrays) to translate and edit data on output for ease of interpretation, and, on input, to convert data from external format to internal format.

READ

5.4.1 Formatted Sequential READ Statement

The formatted sequential READ statement transfers data from the specified logical unit. The characters transmitted are converted to internal format as specified by the format specification. The resulting values are assigned to the elements of the I/O list.

The formatted sequential READ statement has the forms:

```
READ  f[,list]

READ  (u,f[,END=s][,ERR=s])[list]
```

u is a logical unit number.

f is a format specifier.

s is an executable statement label.

list is an I/O list.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant or alphanumeric literal, input data will be read and stored directly into the format specification. For example, the statements

```
      READ (5,100)
100   FORMAT (5HΔDATA)
```

cause five characters to be read and stored in the Hollerith format descriptor. If the characters were HELLO, statement 100 would become:

```
100  FORMAT (5HHELLO)
```

A statement of the form:

```
READ 200, ALPHA,BETA,GAMMA
```

causes data to be read from a system dependent logical unit.

If the number of elements in the I/O list is less than the number of fields in the input record, the excess portion of the record is discarded. If the number of list elements exceeds the number of input fields, an error condition results unless the format specifications state that one or more additional records are to be read (see Sections 6.4 and 6.8).

If no I/O list is present, data transfer is between the record and the format specification.

Examples

```
      READ (1,300) ARRAY
300   FORMAT (20F8.2)
```

(Read a record from logical unit 1, assign fields to ARRAY.)

```
      READ (5,50)
50    FORMAT (25HΔPAGEΔHEADINGΔGOESΔHEREΔΔ)
```

(Read 25 characters from logical unit 5, place them in the FORMAT statement.)

WRITE

5.4.2 Formatted Sequential WRITE Statement

The formatted sequential WRITE statement transfers data to the specified logical unit. The I/O list specifies a sequence of values which are converted to characters and positioned as specified by the format specification.

The formatted sequential WRITE statement has the form:

```
WRITE  (u,f[,ERR=s])[list]
```

u is a logical unit number.

f is a format specifier.

s is an executable statement label.

list is an I/O list.

If no I/O list is present, data transfer is entirely between the record and the format specification.

The data transmitted by a formatted sequential WRITE statement normally constitutes one formatted record. The format specification can, however, specify that additional records are to be written during the execution of that same WRITE statement.

Numeric data output under format control is rounded during the conversion to external format. (If such data is subsequently input for additional calculations, loss of precision may result. In this case, unformatted output is preferable to formatted output.)

The records transmitted by a formatted WRITE statement must not exceed the length that can be accepted by the specified device. For example, a line printer typically cannot print a record that is longer than 132 characters.

Examples

```
     WRITE (6, 650)                  (Output the contents of the
650  FORMAT ('ΔHELLO,ΔTHERE')        FORMAT statement to logical unit
                                     6.)

     WRITE (1,95) AYE,BEE,CEE        (Write one record of three
95   FORMAT (F8.5,F8.5,F8.5)         fields to logical unit 1.)

     WRITE (1,950) AYE,BEE,CEE       (Write three separate records
950  FORMAT (F8.5)                   of one field each to logical
                                     unit 1.)
```

In the last example, format control arrives at the rightmost parenthesis of the FORMAT statement before all elements of the I/O list have been output. Each time this occurs, the current record is terminated and a new record is initiated. Thus, three separate records are written. (See Section 6.7.)

ACCEPT

5.4.3 Formatted ACCEPT Statement

The function of the ACCEPT statement is identical to that of the formatted READ statement, except that input is read from a logical unit normally connected to the terminal keyboard.

The ACCEPT statement has the form:

```
ACCEPT f[,list]
```

f is a format specifier.

list is an I/O list.

The rules for the format reference and I/O list of an ACCEPT statement are the same as those for the formatted READ statement (Section 5.4.1).

Examples

```
     ACCEPT 100, NUMBER              (Accept one Integer value from
100  FORMAT (I4)                     terminal keyboard.)

     ACCEPT 200                      (Read 13 characters from
200  FORMAT ('PUTΔDATAΔHERE')        keyboard, place them in the FORMAT
                                     statement.)
```

TYPE

5.4.4 Formatted TYPE Statement

The TYPE statement functions identically to the formatted WRITE statement except that output is directed to a logical unit normally connected to the terminal printer.

The TYPE statement has the form:

```
TYPE f[,list]
```

f is a format specifier.

list is an I/O list.

The rules for the format reference and I/O list of a TYPE statement are the same as those for the formatted WRITE statement (Section 5.4.2).

Examples

```
      TYPE FMT, BOLD                    (Display the contents of
                                        BOLD on terminal in the
                                        format specified by contents
                                        of array FMT. See Section
                                        5.4)

      TYPE 400                          (Type message from FORMAT
400   FORMAT ('ΔMOUNTΔNEWΔTAPEΔREEL')   statement on terminal.)
```

PRINT

5.4.5 Formatted PRINT Statement

The function of the PRINT statement is the same as that of the formatted WRITE statement and TYPE statement, except that output is directed to a logical unit normally connected to a line printer.

The PRINT statement has the form:

```
PRINT f[,list]
```

f is a format specifier.

list is an I/O list.

The format reference and I/O list in a PRINT statement follow the same rules as specified for the formatted sequential WRITE statement (Section 5.4.2).

Examples

```
      PRINT 999, NPAGE              (Print the page number in
999   FORMAT (1H1,100X,'PAGEΔΔ',I3) upper right-hand corner of new
                                    page.)

      PRINT 222                     (Print the contents of FORMAT
222   FORMAT ('ΔENDΔOFΔLISTING')    statement on line printer.)
```

5.5 UNFORMATTED DIRECT ACCESS INPUT/OUTPUT

Unformatted direct access READ and WRITE statements are used to perform direct access I/O with a file on a direct access device. The DEFINE FILE or OPEN statement is used to establish the number of records, and the size of each record, in a file to which direct access I/O is to be performed. Each direct access READ or WRITE statement contains an integer expression that specifies the number of the record to be accessed. The record number must not be less than one nor greater than the number of records defined for the file.

In FORTRAN IV-PLUS the expression that specifies the record number can be of any type. It is converted to integer type if necessary.

READ

5.5.1 Unformatted Direct Access READ Statement

The unformatted direct access READ statement positions the input file to a specified record and transfers the fields in that record to the elements in the I/O list without translation.

The unformatted direct access READ statement has the form:

```
READ (u'r[,ERR=s]) [list]
```

u is a logical unit number.

r is the record number.

s is an executable statement label.

list is an I/O list.

If there are more fields in the input record than elements in the I/O list, the excess portion of the record is discarded. If there is insufficient data in the record to satisfy the requirements of the I/O list, an error condition results.

The unit number in the unformatted direct access READ statement must refer to a unit that has been previously defined for direct access processing.

Examples

```
READ (1'10) LIST(1),LIST(8)      (Read record 10 of a file on
                                  logical unit 1, assign two
                                  Integer values to specified
                                  elements of array LIST.)

READ (4'58) (RHO(N),N=1,5)       (Read record 58 of a file on
                                  logical unit 4, assign five
                                  Real values to array RHO.)
```

WRITE

5.5.2 Unformatted Direct Access WRITE Statement

The unformatted direct access WRITE statement transmits the values of the elements in the I/O list to a particular record of a direct access file. The data is written in internal format without translation.

The unformatted direct access WRITE statement has the form:

```
WRITE (u'r[,ERR=s]) [list]
```

u is a logical unit number.

r is the record number.

s is an executable statement label.

list is an I/O list.

If the amount of data to be transmitted exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, the contents of the unused portion of the record are zero-filled.

Examples

`WRITE (2'35) (NUM(K),K=1,10)`	(Output ten Integer values to record 35 of the file connected to logical unit 2.)
`WRITE (3'J) ARRAY`	(Output the entire contents of ARRAY to the file connected to logical unit 3 into the record indicated by the value of J.)

5.6 FORMATTED DIRECT ACCESS INPUT/OUTPUT

Formatted direct access READ and WRITE statements are used to perform direct access I/O of character data with a file on a direct access device. The OPEN statement is used to establish the attributes of the file. Each READ or WRITE contains an expression that specifies the number of the record to be accessed. The record number must not be less than one nor greater than the number of records defined for the file.

READ

5.6.1 Formatted Direct Access READ Statement

The formatted direct access READ statement inputs the specified record from the direct access file currently connected to the unit. The characters in the record are converted as specified by the format specification. The resulting values are assigned to the elements specified by the list.

The formatted direct access READ statement has the form:

```
READ (u'r,f[,ERR=s])[list]
```

u is a logical unit number.

r is the record number.

f is a format specifier.

s is an executable statement label.

list is an I/O list.

If the I/O list and format specify more characters than a record contains, or specify additional records, an error condition results.

WRITE

5.6.2 Formatted Direct Access WRITE Statement

The formatted direct access WRITE statement outputs the specified record to the direct access file that is currently connected to the unit. The list specifies a sequence of values which are converted to characters and positioned as specified by the format specification.

The formatted direct access WRITE statement has the form:

```
WRITE (u'r,f[,ERR=s])[list]
```

u is a logical unit number.

r is the record number.

f is a format specifier.

s is an executable statement label.

list is an I/O list.

If the values specified by the list and format do not fill the record, space characters are appended to fill the record.

If the I/O list and format specify more characters than can fit into a record, or specify additional records, an error condition results.

5.7 LIST-DIRECTED INPUT/OUTPUT

List-directed input and output statements provide a method for obtaining simple sequential formatted input or output without the need for FORMAT statements. On input, values are read from the unit, converted to internal format, and assigned to the elements of the I/O list. On output, values in the I/O list are converted to characters and written in a fixed format according to the data type of the value. The I/O list must be present. Records written by list-directed output statements are suitable for input by list-directed input statements provided they do not contain alphanumeric literals or Hollerith constants.

Both formatted and list-directed I/O statements can refer to the same unit. All operations permitted for formatted sequential I/O are supported with list-directed I/O statements; however, backspacing over list-directed records leaves the file position undefined. When files are read that contain both formatted and list-directed records, the user program must assure that each record is read properly.

READ

5.7.1 List-Directed READ Statement

The list-directed READ statement transfers data from the specified unit, translates from external to internal format, and assigns the input values to the elements of the I/O list in the order in which they appear, from left to right.

The list-directed READ statement has the form:

```
READ *,list

READ (u,*[,END=s][,ERR=s]) list
```

u	is a logical unit number.
*	indicates list-directed formatting.
s	is an executable statement label.
list	is an I/O list.

The external record contains a sequence of values and value separators.

A value can be:

1. A constant

   Each input constant has the form of the corresponding FORTRAN constant. A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur between the open parenthesis and the first constant, around the separating comma and between the second constant and the close parenthesis. A logical constant is either T for .TRUE. or F for .FALSE. Hollerith, octal, and alphanumeric constants are not permitted.

2. A null value

   A null value is specified by two consecutive commas with no intervening constant. Spaces can be embedded between the commas. A null value specifies that the corresponding list element is to remain unchanged. A null value cannot be used for either part of a complex constant, but can represent an entire complex constant.

3. A repetition of constants in the form r*c

   The form r*c indicates r occurrences of c where r is a non-zero, unsigned integer constant and c is a constant. Spaces are not permitted except within the constant c as specified above.

4. A repetition of null values in the form r*

   The form r* indicates r occurrences of null where r is an unsigned integer constant.

A value separator can be:

1. one or more spaces or tabs
2. a comma with or without surrounding spaces or tabs
3. a slash with or without surrounding spaces or tabs

   A slash separator causes termination of processing on the input statement and record; all remaining I/O list elements are unchanged.

The end of a record is equivalent to a space character. Spaces at the beginning of a record are ignored.

The types of acceptable input constants are: integer, real, double precision, logical and complex. The data type of the value and the conversion from external to internal form is determined by the form of the constant. If the data types of the list element and the corresponding constant do not match, conversion is performed according to the rules for arithmetic assignment described in Table 3-1.

Each input statement will read one or more records as required to satisfy the I/O list. If all of the values in a record are not used, either because a slash separator is encountered or the I/O list is exhausted, the remaining values in the record are lost.

Example

If the program unit consists of

```
DOUBLE PRECISION T
COMPLEX C,D
LOGICAL L,M
READ (1,*) I,R,C,D,L,M,J,K,S,T,A,B
     .
     .
     .
```

and the record read contains:

```
4Δ6.3Δ(3.4,4.2),Δ(3,Δ2 )Δ,ΔT,F,,3*14.6Δ/
```

The following values are assigned to the I/O list elements:

```
I = 4
R = 6.3
C = (3.4,4.2)
D = (3.0,2.0)
L = .TRUE.
M = .FALSE.
K = 14
S = 14.6
T = 14.6 D0
```

A, B and J will be unchanged.

5.7.2 List-Directed WRITE Statement

The list-directed WRITE statement transmits the elements in the I/O list to the specified unit, translating and editing each value according to the data type of the value.

The list-directed WRITE statement has the form:

```
WRITE (u,*[,ERR=s]) list
```

u	is a logical unit number.
*	indicates list-directed formatting.
s	is an executable statement label.
list	is an I/O list.

The forms of the output values produced are the same as that required for input as described for the list-directed READ statement (Section 5.7.1). In addition, alphanumeric literals and Hollerith constants can be output; these constants are not delimited by apostrophes. The values produced are separated by one space. Each value is transmitted in a default format as illustrated in Table 5-1.

Table 5-1
List-Directed Output Formats

Data Type	Output format
Logical*1	I5
Logical*2	L2
Logical*4	L2
Integer*2	I7
Integer*4	I12
Real*4	1PG15.7
Real*8	1PG25.16
Complex*8	1X,'(',1PG14.7, ',', 1PG14.7,')'
Alphanumeric literals	1X,nAl (n=length of the literal string)

Octal values, null values, slash (/) separators, and repeated forms are not produced. Each output record begins with a space for carriage control. Each output statement writes one or more complete records. Each value is contained within a single record, except for alphanumeric literals which are longer than a record.

Example

The program unit

```
DIMENSION A(5)
DATA A/5*3.4/
WRITE (1,*) 'ARRAY∆VALUES∆FOLLOW'
WRITE (1,*) A,5
```

writes the following records:

```
ARRAY△VALUES△FOLLOW
△△△3.400000△△△△△△△3.400000△△△△△△△3.400000△△△△△△△3.400000
△△△3.400000△△△△△△△△△△5
```

ACCEPT

5.7.3 List-Directed ACCEPT Statement

The list-directed ACCEPT statement functions identically to the list-directed READ statement, except that input is read from a logical unit normally connected to the terminal keyboard.

The list-directed ACCEPT statement has the form:

```
ACCEPT *, list
```

`*` indicates list-directed formatting.

`list` is an I/O list.

The rules for the form of the external records read by the list-directed ACCEPT statement are the same as those for the list-directed READ statement (Section 5.7.1).

Examples

```
ACCEPT *,I,J,K,A(I,J,K)
ACCEPT *,I,(Q(J),J=1,I)
```

TYPE

5.7.4 List-Directed TYPE Statement

The list-directed TYPE statement functions identically to the list-directed WRITE statement, except that output is directed to a logical unit normally connected to the terminal printer.

The list-directed TYPE statement has the form:

```
TYPE *, list
```

`*` indicates list-directed formatting.

`list` is an I/O list.

The form of the output values of a list-directed TYPE statement are the same as those for the list-directed WRITE statement (Section 5.7.2).

Examples

```
TYPE *,'THE△ANSWER△IS',I
TYPE *,(I,XX(I),I=1,10)
```

PRINT

5.7.5 List-Directed PRINT Statement

The list-directed PRINT statement functions identically to the list-directed WRITE statement, except that output is directed to a logical unit normally connected to a line printer.

The list-directed PRINT statement has the form:

```
PRINT *, list
```

`*` indicates list-directed formatting.

`list` is an I/O list.

The form of the output values of a list-directed PRINT statement are the same as those for the list-directed WRITE statement (Section 5.7.2).

Examples

```
PRINT *,((QQ(M,N),M=1,100),N=1,100)
PRINT *,'THEΔARRAYΔZΔIS',Z
```

5.8 TRANSFER OF CONTROL ON END-OF-FILE OR ERROR CONDITIONS

Any type of READ or WRITE statement can contain a specification that control is to be transferred to another statement if the I/O statement encounters an error condition or the end of the file. These specifications have the form:

```
END=s
```

and

```
ERR=s
```

`s` is the label of an executable statement to which control is to be transferred.

A READ or WRITE statement can contain either or both of the above specifications, in either order. Any such specification must follow the unit number, record number, and/or format specification.

If an end-of-file condition is encountered during an I/O operation, the READ statement transfers control to the statement named in the END=s specification. If no such specification is present, an error condition results. An end-of-file condition occurs when no more records exist in a sequential file, or when an end-of-file record produced by the ENDFILE statement is read.

If a READ or WRITE statement encounters an error condition during an I/O operation, it transfers control to the statement whose label appears in the ERR=s specification. If no ERR=s specification is present, the I/O error causes program execution to terminate.

The statement label in the END=s or ERR=s specification must refer to an executable statement that exists within the same program unit as the I/O statement.

Examples of I/O statements containing END=s and ERR=s specifications follow:

```
READ (8,END=550) (MATRIX(K),K=1,100)    (Pass control to statement
                                         550 when end-of-file is
                                         encountered on logical unit
                                         8.)

WRITE (5,50,ERR=390)                     (Pass control to statement
                                         390 on error.)

READ (1'INDEX,ERR=150) ARRAY             (Pass control to statement
                                         150 on error.)
```

NOTE

An end-of-file condition can not occur during direct access READ or WRITE statements. Attempting to READ or WRITE a record using a record number greater than the maximum specified for the unit is an error condition.

The FORTRAN User's Guide describes system subroutines that can control processing of error conditions and obtain information from the I/O system concerning the type of error condition that has occurred.

5.9 AUXILIARY INPUT/OUTPUT STATEMENTS

The statements in this category are used to perform file management functions.

REWIND

5.9.1 REWIND Statement

The REWIND statement causes a currently open sequential file to be repositioned to the beginning of the file.

The REWIND statement has the form:

```
REWIND u
```

u is a logical unit number.

The unit number in the REWIND statement must refer to a directory-structured device (e.g., disk). A file must be open on that device.

Example

```
REWIND 3     (Reposition logical unit 3 to beginning of
              currently open file.)
```

BACKSPACE

5.9.2 BACKSPACE Statement

The BACKSPACE statement repositions a currently open sequential file backward one record and repositions to the beginning of that record. On the execution of the next I/O statement for that unit, that record is available for processing.

The BACKSPACE statement has the form:

```
BACKSPACE u
```

u is a logical unit number.

The unit number must refer to a directory-structured device (e.g., disk). A file must be open on that device.

Example

```
BACKSPACE 4    (Reposition open file on logical unit 4 to
                beginning of the previous record.)
```

ENDFILE

5.9.3 ENDFILE Statement

The ENDFILE statement writes an end-file record to the specified sequential unit.

The ENDFILE statement has the form:

```
ENDFILE u
```

u is a logical unit number.

Example

```
ENDFILE 2    (Output an end-file record to logical unit 2.)
```

DEFINE FILE

5.9.4 DEFINE FILE Statement

The DEFINE FILE statement establishes the size and structure of a file upon which direct access I/O is to be performed.

The DEFINE FILE statement has the form:

```
DEFINE FILE u (m,n,U,v) [,u(m,n,U,v)]...
```

u is an integer constant or integer variable that specifies the logical unit number.

m is an integer constant or integer variable that specifies the number of records in the file.

n is an integer constant or integer variable that specifies the length, in words, of each record.

U specifies that the file is unformatted (binary). The letter U is the only acceptable entry in this position.

v is an integer variable, called the associated variable of the file. At the conclusion of each direct access I/O operation the record number of the next higher numbered record in the file is assigned to v.

The DEFINE FILE statement specifies that a file containing m fixed-length records of n words each exists, or is to exist, on logical unit u. The records in the file are sequentially numbered from 1 through m.

The DEFINE FILE statement must be executed before the first direct access I/O statement that refers to the specified file.

The DEFINE FILE statement also establishes the integer variable v as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in v the record number of the record immediately following the one just read or written. Since the associated variable always points to the next sequential record in the file (unless it is redefined by an assignment, input or FIND statement), direct access I/O statements can be used to perform sequential processing of the file, by using the associated variable of the file as the record number specifier.

If the file is to be processed by more than one program unit, or in an overlay environment, the associated variable should be placed in a resident named COMMON block.

In FORTRAN IV-PLUS the parameter m can be an INTEGER*4 quantity.

Example

```
DEFINE FILE 3 (1000,48,U,NREC)
```

This statement specifies that logical unit 3 is to be connected to a file of 1000 fixed-length records, each record of which is 48 words long. The records are numbered sequentially from 1 through 1000, and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the one just processed.

FIND

5.9.5 FIND Statement

The FIND statement positions a direct access file on a specified unit to a particular record and sets the associated variable of the file to that record number. No data transfer takes place.

The FIND statement has the form:

```
FIND (u'r)
```

u is a logical unit number.

r is the record number.

The unit number in the statement must refer to a unit that must have been previously defined for direct access processing.

The record number must not be less than 1 nor greater than the number of records defined for the file.

Examples

```
FIND (1'1)        (Position logical unit 1, and its associated
                  variable, to the first record of the file.)

FIND (4'INDX)     (Position the file and associated variable
                  to record identified by contents of INDX.)
```

OPEN

5.9.6 OPEN Statement

An OPEN statement can be used to connect an existing file to a logical unit, or to create a new file and connect it to a logical unit. In addition, the statement can contain specifications for file attributes that will direct the creation and/or subsequent processing.

The OPEN statement has the form:

```
OPEN (p[,p]...)
```

p is a specification in one of the following forms:

key	key is a keyword.
key=e	e is a numeric expression.
key=s	s is an executable statement label.
key=lit	lit is an alphanumeric literal of special significance.
key=v	v is an integer variable name.
key=n	n is an array name, variable name, array element name, or alphanumeric literal.

The keywords available for attribute specifications are summarized in Table 5-2, and described in detail in the following sections.

Attribute specifications can appear in any order. In most cases, attribute specifications are optional and if not present, default specifications will be provided.

In the discussions that follow, a numeric expression can be any integer, real or double precision expression. The value of the expression will be converted to integer type prior to use in the OPEN statement.

NOTE

All OPEN statement keywords and options are accepted by all PDP-11 FORTRAN processors. Some keywords and options may not be supported on a specific operating system. Consult the appropriate FORTRAN User's Guide for information on system-specific options.

Table 5-2
Keywords in the OPEN Statement

KEYWORD	FUNCTION	VALUES
UNIT	logical unit number	e
NAME	file specification	n
TYPE	file type	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'
ACCESS	access method	'SEQUENTIAL' 'DIRECT' 'APPEND'
READONLY	read-only file access	
FORM	file format	'FORMATTED' 'UNFORMATTED'
RECORDSIZE	direct access record length	e
ERR	error condition transfer label	s
BUFFERCOUNT	number of buffers	e
INITIALSIZE	file allocation size	e
EXTENDSIZE	file extension increment	e
NOSPANBLOCKS	unspanned records	
SHARED	shared file access	
DISPOSE DISP	file disposition	'SAVE' 'KEEP' 'PRINT' 'DELETE'
ASSOCIATEVARIABLE	associated variable name	v
CARRIAGECONTROL	carriage control type	'FORTRAN' 'LIST' 'NONE'
MAXREC	number of direct access records	e
BLOCKSIZE	physical block size	e

e is a numeric expression.
n is a variable name, array name, array element name, or alphanumeric literal.
s is an executable statement label.
v is an integer variable name.

5.9.6.1 UNIT Keyword

The form

UNIT = u

specifies the logical unit to which a file is to be connected; u is a numeric expression. The unit specification must appear in the list. There must not be a file connected to the logical unit at the time the OPEN statement is executed.

5.9.6.2 NAME Keyword

The form

NAME = fln

specifies the name of the file to be connected to the unit; fln may be an alphanumeric literal, variable name, array name or array element name. The name can be any file specification acceptable to the operating system. Default file name conventions are described in the appropriate FORTRAN User's Guide.

If the file name is stored in a variable or array, the name must be terminated by an ASCII null character (zero byte).

5.9.6.3 TYPE Keyword

The form

TYPE = typ

specifies the type of the file to be opened; typ is a literal of the form 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. If 'OLD' is specified, the file must already exist. If 'NEW' is specified, a new file will be created. If 'SCRATCH' is specified a new file is created and connected to the specified unit for use by the executable program; the file is deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. If 'UNKNOWN' is specified the processor will first try 'OLD' and if the file is not found it will then try 'NEW', thereby creating a new file. The default is 'NEW'.

5.9.6.4 ACCESS Keyword

The form

ACCESS = acc

specifies whether the file is direct access or sequential; acc is a literal of the form: 'DIRECT', 'SEQUENTIAL', or 'APPEND'. 'DIRECT' specifies a direct access file. 'SEQUENTIAL' specifies a sequential file. 'APPEND' implies a sequential file and positioning after the last record of the file. The default is 'SEQUENTIAL'.

5.9.6.5 READONLY Keyword

The form

READONLY

specifies that an existing file is to be read and prohibits writing to that file.

5.9.6.6 FORM Keyword

The form

FORM = ft

specifies whether the file being opened is to be read and written using formatted or unformatted READ or WRITE statements; ft is a literal of the form 'FORMATTED' or 'UNFORMATTED'. For sequential files, 'FORMATTED' is the default. For direct access files, 'UNFORMATTED' is the default.

5.9.6.7 RECORDSIZE Keyword

The form

RECORDSIZE = rl

specifies the length of each logical record of a direct access file; rl is a numeric expression. If the records are formatted, the length is the number of characters; if the records are unformatted, the length is the number of storage units (double words).

5.9.6.8 ERR Keyword

The form

ERR = s

specifies a transfer on error condition; s is an executable statement label. The ERR= option as specified in the OPEN statement applies only to that OPEN and not to subsequent I/O operations on the unit. If an error condition occurs, no file is opened or created.

5.9.6.9 BUFFERCOUNT Keyword

The form

BUFFERCOUNT = bc

specifies the number of buffers to be associated with the unit for multi-buffered I/O; bc is a numeric expression. If not specified, or zero, the system default (normally one) is assumed.

5.9.6.10 INITIALSIZE Keyword

The form

INITIALSIZE = is

specifies the number of blocks in the initial allocation of space for a new file on a disk unit; is is a numeric expression. This keyword can be used only for new files and only for allocation to a disk unit. If zero or not present, the system default initial allocation will be used.

5.9.6.11 EXTENDSIZE Keyword

The form

EXTENDSIZE = es

specifies the number of blocks by which to extend a file when additional file storage must be allocated; es is a numeric expression. If zero or not present, the system default will be used.

5.9.6.12 NOSPANBLOCKS Keyword

The form

NOSPANBLOCKS

specifies that records are not to cross disk block boundaries. If any record exceeds the size of a physical block, an error condition results.

5.9.6.13 SHARED Keyword

The form

SHARED

specifies that the file is to be opened for shared access by more than one program executing simultaneously. Consult the appropriate FORTRAN User's Guide for additional information on the implications of this specifier.

5.9.6.14 DISPOSE Keyword

The forms

```
DISPOSE = dis
DISP    = dis
```

determine the disposition of the file when the unit is closed; dis is a literal of the form 'SAVE', 'KEEP', 'PRINT', or 'DELETE'. The default value is 'SAVE'. 'SAVE' causes the file to be retained after the unit is closed; 'KEEP' is a synonym for 'SAVE'. 'DELETE' causes the file to be deleted when the unit is closed. 'PRINT' causes the file to be printed on the system line printer. On some systems, 'PRINT' implies deletion after printing; consult the appropriate FORTRAN User's Guide for further information. A read-only file cannot be printed or deleted. A scratch file cannot be printed or saved.

5.9.6.15 ASSOCIATEVARIABLE Keyword

The form

```
ASSOCIATEVARIABLE = asv
```

specifies the integer variable asv, that, at the conclusion of each direct access I/O operation, contains the record number of the next sequential record in the file. This specifier is ignored for a sequential file.

5.9.6.16 CARRIAGECONTROL Keyword

The form

```
CARRIAGECONTROL = cc
```

determines the kind of carriage control processing to be used when printing a file; cc is a literal of the form 'FORTRAN', 'LIST', or 'NONE'. The default for formatted files is 'FORTRAN'; for unformatted files, 'NONE'. 'FORTRAN' specifies normal FORTRAN interpretation of the first character; 'LIST' specifies single spacing between records and 'NONE' specifies no implied carriage control.

5.9.6.17 MAXREC Keyword

The form

```
MAXREC = mr
```

specifies the maximum number of records to be permitted in a direct access file; mr is a numeric expression. The default is no maximum number of records. This specifier is ignored for a sequential file.

5.9.6.18 BLOCKSIZE Keyword

The form

```
BLOCKSIZE = bks
```

specifies the physical block size to be used for the unit; bks is a numeric expression. This specifier only has effect for magnetic tape files; it has no effect for disk files. The default is the system default for the device.

5.9.6.19 OPEN Statement Examples

Examples

```
OPEN (UNIT=3,TYPE='SCRATCH',ACCESS='DIRECT',
      INITIALSIZE=50,RECORDSIZE=64)
```

Create a 50 block direct access file for temporary storage. The file is deleted at program termination.

```
OPEN (UNIT=I,NAME='MT0:DATA.DAT',BLOCKSIZE=8192,
      TYPE='NEW',ERR=14)
```

Create a file on magnetic tape with a large blocksize for efficient processing.

```
OPEN (UNIT=I,NAME='MT0:DATA.DAT',READONLY,
      TYPE='OLD',BLOCKSIZE=8192)
```

Open the file created in the previous example for input.

CLOSE

5.9.7 CLOSE Statement

The CLOSE statement has the form:

```
CLOSE (UNIT=u [, {DISPOSE} =p] [,ERR=s ] )
              {DISP    }
```

u is a logical unit number.

p is a literal that determines the disposition of the file. Its values are 'SAVE', 'KEEP', 'DELETE', and 'PRINT'. 'SAVE' and 'KEEP' are synonyms; if either is specified, the file is retained after the CLOSE. If 'DELETE' is specified the file ceases to exist after the CLOSE. 'PRINT' causes the file to be printed by the system line printer. For scratch files the default is 'DELETE'; for all other files the default is 'SAVE'.

s is an executable statement label.

The CLOSE statement disconnects a file from a unit. The disposition specified in a CLOSE statement supersedes the disposition specified in the OPEN statement, except that a file opened as a scratch file can not be saved or printed nor can a file opened for read-only access be printed or deleted.

Examples

```
CLOSE(UNIT=1, DISPOSE='PRINT')
```

Close the file on unit 1 and submit the file for printing.

```
CLOSE(UNIT=J,DISPOSE='DELETE',ERR=99)
```

Close the file on unit J and delete it.

ENCODE
DECODE

5.10 ENCODE AND DECODE STATEMENTS

These two statements perform data transfers according to format specifications, translating data from internal format to character format, or vice versa. Unlike conventional formatted I/O statements, however, these data transfers take place entirely between variables or arrays in the FORTRAN program.

The ENCODE and DECODE statements are written as follows:

```
ENCODE(c,f,b[,ERR=s])[list]

DECODE(c,f,b[,ERR=s])[list]
```

c is an integer expression representing the number of characters (bytes) that are to be converted or that are to result from the conversion. (This is analogous to the length of an external record.)

f is a format specifier. If the format specifies more than one record, an error condition results.

b is the name of an array, array element or variable. In the ENCODE statement, this entity receives the characters. In the DECODE statement, it contains the characters that are to be translated to internal format. b must not be the name of a VIRTUAL array or a VIRTUAL array element.

s is an executable statement label.

list is an I/O list. In the ENCODE statement, the I/O list contains the data that is to be converted to characters. In the DECODE statement, the list receives the data that has been translated from characters to internal format.

The ENCODE statement causes the I/O list elements to be translated to character format according to the format specification and stored in the entity b, in an analogous fashion to a WRITE statement.

The DECODE statement causes character data in the entity b to be interpreted and converted according to the format specification and assigned to the I/O list elements; this processing is analogous to a READ statement.

If the entity b is an array, the elements of that array are processed in the order of subscript progression.

The number of characters that can be processed by the ENCODE or DECODE statement is dependent on the data type of the entity b in that statement. An INTEGER*2 array, for example, can contain two characters per element, so the maximum number of characters is twice the number of elements in that array.

The interaction between format control and the I/O list is the same as for a formatted I/O statement.

Example

```
      DIMENSION A(3),K(3)
      DATA A /'1234','5678','9012'/
      DECODE (12,100,A) K
  100 FORMAT (3I4)
```

Execution of the DECODE statement causes the 12 characters in array A to be converted to Integer format (specified by statement 100) and stored in array K, as follows:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

CHAPTER 6

FORMAT STATEMENTS

6.1 OVERVIEW

FORMAT statements are nonexecutable statements used in conjunction with formatted I/O statements and with ENCODE and DECODE statements. The FORMAT statement describes the format in which data fields are transmitted, and the data conversion and editing to be performed to achieve that format.

FORMAT

The FORMAT statement has the form:

FORMAT ($q_1 f_1 s_1 f_2 s_2 \cdots f_n q_n$)

where each f is a field descriptor, or a group of field descriptors enclosed in parentheses, each s is a field separator and each q is zero or more slash (/) record terminators. The entire list of field descriptors and field separators including the parentheses is called the format specification. The list must be enclosed in parentheses.

A field descriptor in a format specification has the form:

[r]Cw[.d]

- r represents a repeat count which specifies that the field descriptor is to be applied to r successive fields. If the repeat count is omitted, it is assumed to be 1. See Section 6.2.17 for further discussion of field repetition.
- C is a format code.
- w specifies the field width.
- d specifies the number of characters to the right of the decimal point.

The terms r, w, and d must all be unsigned integer constants less than or equal to 255.

The field separators are comma and slash. A slash has the additional function of being a record terminator. The functions of the field separators are described in detail in Section 6.4.

The field descriptors used in format specifications are as follows:

1. Integer: Iw, Ow

2. Logical: Lw

3. Real, Double Precision, Complex: Fw.d, Ew.d, Dw.d, Gw.d

4. Literal and editing: Aw, nH, '...', nX, Tn, Q, $, :

(In the alphanumeric and editing field descriptors, n specifies a number of characters or character positions.)

Any of the F, E, D, or G field descriptors can be preceded by a scale factor of the form:

nP

where n is an optionally signed integer constant in the range -127 to +127 that specifies the number of positions the decimal point is to be scaled to the left or right. The scale factor is described in Section 6.2.15.

During data transmission, the format specification is scanned from left to right. Data conversion is performed by correlating the values in the I/O list with the corresponding field descriptors. In the case of H field descriptors and alphanumeric literals, data transmission takes place entirely between the field descriptor and the external record. The interaction between the format specification and the I/O list is described in detail in Section 6.7.

6.2 FIELD DESCRIPTORS

The individual field descriptors that can appear in a format specification are described in detail in the following sections. The field descriptors ignore leading spaces in the external field, but treat embedded and trailing spaces as zeros.

6.2.1 I Field Descriptor

The I field descriptor governs the translation of integer data. It has the form:

Iw

The I field descriptor causes an input statement to read w characters from the external record and to assign them as an integer value to the corresponding integer element of the I/O list. The external data must be an integer; it must not contain a decimal point or exponent field. The I field descriptor interprets an all-blank field as a zero value. If the value of the external field exceeds the range of the corresponding integer list element, an error occurs. If the first non-blank character of the external field is a minus symbol, the I field descriptor causes the field to be stored as a negative value; a field preceded by a plus symbol, or an unsigned field, is treated as a positive value. For example:

Format	External Field	Internal Representation
I4	2788	2788
I3	-26	-26
I9	ΔΔΔΔΔΔ312	312
I9	3.12	not permitted; error

The I field descriptor causes an output statement to transmit the value of the corresponding integer I/O list element to an external field w characters in length, right justified, replacing any leading zeros with spaces. If the value of the list element is negative, the field will have a minus symbol as its leftmost non-blank character. Space must therefore be included in w for a minus symbol if any are expected. Plus symbols, on the other hand, are suppressed and need not be accounted for in w. If w is too small to contain the output value, the entire external field is filled with asterisks. For example:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	ΔΔ174
I2	3244	**
I3	-473	***
I7	29.812	not permitted; error

6.2.2 O Field Descriptor

The O field descriptor governs the transmission of octal values. It has the form:

Ow

The O field descriptor causes an input statement to read w characters from the external record and to assign them as an octal value to the corresponding I/O list element. The list element must be of integer or logical type. The external field must contain only the numerals 0 through 7; it must not contain a sign, a decimal point, or an exponent field. For example:

Format	External Field	Internal Octal Representation
O5	32767	32767
O4	16234	1623
O6	13ΔΔΔΔ	130000
O3	97Δ	not permitted; error

The O field descriptor causes an output statement to transmit the value of the corresponding I/O list element, right justified, to a field w characters long. If the data does not fill the field, leading spaces are inserted; if the data exceeds the field width, the entire field is filled with asterisks. No signs are output; a negative value is transmitted in its octal (two's complement) form. The I/O list element must be of integer or logical type. For example:

Format	Internal (Decimal) Value	External Representation
O6	32767	Δ77777
O6	-32767	100001
O2	14261	**
O4	27	ΔΔ33
O5	13.52	not permitted; error

6.2.3 F Field Descriptor

The F field descriptor specifies the data conversion and editing of real or double precision values, or the real or imaginary parts of complex values. It has the form:

Fw.d

On input, the F field descriptor causes w characters to be read from the external record and to be assigned as a real value to the corresponding I/O list element. If the first non-blank character of the external field is a minus sign, the field is treated as a negative value; a field that is preceded by a plus sign, or an unsigned field, is considered to be positive. An all-blank field is considered to have a value of zero. In all appearances of the F field descriptor, w must be greater than or equal to d+1.

If the field contains neither a decimal point nor an exponent, it is treated as a real number of w digits, in which the rightmost d digits are to the right of the decimal point. If the field contains an explicit decimal point, the location of that decimal point overrides the location specified by the field descriptor. If the field contains an exponent (in the same form as described in Section 2.4.2 for real constants or Section 2.4.3 for double precision constants), that exponent is used in establishing the magnitude of the value before it is assigned to the list element. For example:

Format	External Field	Internal Representation
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

On output, the F field descriptor causes the value of the corresponding I/O list element to be rounded to d decimal positions and transmitted to an external field w characters in length, right justified. If the converted data consists of fewer than w characters, leading spaces are inserted; if the data exceeds w characters, the entire field is filled with asterisks.

The total field width specified must be large enough to accommodate a minus sign, if any are expected (plus signs are suppressed), at least one digit to the left of the decimal point, the decimal point itself, and d digits to the right of the decimal. For this reason, w should always be greater than or equal to (d+3). Examples follow:

Format	Internal Value	External Representation
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.3	51.44	**
F10.4	-23.24352	ΔΔ-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

6.2.4 E Field Descriptor

The E field descriptor specifies the transmission of real or double precision values in exponential format. It has the form:

Ew.d

The E field descriptor causes an input statement to input w characters from the external record. It interprets and assigns that data in exactly the same way as the F field descriptor. For example:

Format	External Field	Internal Representation
E9.3	734.432E3	734432.0
E12.4	ΔΔ1022.43E-6	1022.43E-6
E15.3	52.3759663ΔΔΔΔΔ	52.3759663
E12.5	210.5271D+10	210.5271E10

Note that in the last example the E field descriptor disregards the double precision connotation of a D exponent field indicator and treats it as though it were an E indicator.

The E field descriptor causes an output statement to transmit the value of the corresponding list element to an external field w characters in width, right justified. If the number of characters in the converted data is less than w, leading spaces are inserted; if the number of characters exceeds w, the entire field is filled with asterisks. The corresponding I/O list element must be of real, double precision, or complex type.

Data output under control of the E field descriptor is transmitted in a standard form, consisting of a minus sign if the value is negative (plus signs are suppressed), a zero, a decimal point, d digits to the right of the decimal, and a 4-character exponent of the form:

E+nn

or

E-nn

where nn is a 2-digit integer constant. The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

Because w must be large enough to include a minus sign (if any are expected), a zero, a decimal point, and an exponent, in addition to d digits, w should always be equal to or greater than (d+7). Some examples are:

Format	Internal Value	External Representation
E9.2	475867.222	Δ0.48E+06
E12.5	475867.222	Δ0.47587E+06
E12.3	0.00069	ΔΔΔ0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****

6.2.5 D Field Descriptor

The D field descriptor specifies the transmission of real or double precision values. It has the form:

Dw.d

On input, the D field descriptor functions exactly as an equivalent E field descriptor, except that the input data is converted and assigned as a double precision entity, as in the following examples:

Format	External Field	Internal Representation
D10.2	12345ΔΔΔΔΔ	12345000.0D0
D10.2	ΔΔ123.45ΔΔ	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

On output the effect of the D field descriptor is identical to that of the E field descriptor, except that the D exponent field indicator is used in place of the E indicator. For example:

Format	Internal Value	External Representation
D14.3	0.0363	ΔΔΔΔΔ0.363D-01
D23.12	5413.87625793	ΔΔΔΔΔ0.541387625793D+04
D9.6	1.2	*********

6.2.6 G Field Descriptor

The G field descriptor transmits real, double precision, or complex data in a form that is in effect a combination of the F and E field descriptors. It has the form:

Gw.d

On input, the G field descriptor functions identically to the F field descriptor (see Section 6.2.3).

On output, the G field descriptor causes the value of the corresponding I/O list element to be transmitted to an external field w characters in length, right justified. The form in which the value is output is a function of the magnitude of the value, as described in Table 6-1.

Table 6-1
Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
m < 0.1	Ew.d
0.1 ≤ m < 1.0	F(w-4).d, 'ΔΔΔΔ'
1.0 ≤ m < 10.0	F(w-4).(d-1), 'ΔΔΔΔ'
.	.
.	.
.	.
10d-2 ≤ m < 10d-1	F(w-4).1, 'ΔΔΔΔ'
10d-1 ≤ m < 10d	F(w-4).0, 'ΔΔΔΔ'
m ≥ 10d	Ew.d

The 'ΔΔΔΔ' field descriptor, which is (in effect) inserted by the G field descriptor for values within its range, specifies that four spaces are to follow the numeric data representation.

The field width, w, must include space for a minus sign, if any are expected (plus signs are suppressed), at least one digit to the left of the decimal point, the decimal point itself, d digits to the right of the decimal, and (for values that are outside the effective range of the G field descriptor) a 4-character exponent. Therefore, w should always be equal to or greater than (d+7). Examples of G output conversions are:

Format	Internal Value	External Representation
G13.6	0.01234567	Δ0.123457E-01
G13.6	-0.12345678	-0.123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457ΔΔΔΔ
G13.6	12.34567890	ΔΔ12.3457ΔΔΔΔ
G13.6	123.45678901	ΔΔ123.457ΔΔΔΔ
G13.6	-1234.56789012	Δ-1234.57ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457.ΔΔΔΔ
G13.6	-1234567.89012345	-0.123457E+07

For comparison, consider the following example of the same values output under the control of an equivalent F field descriptor.

Format	Internal Values	External Representation
F13.6	0.01234567	ΔΔΔΔΔ0.012346
F13.6	-0.12345678	ΔΔΔΔ-0.123457
F13.6	1.23456789	ΔΔΔΔΔ1.234568
F13.6	12.34567890	ΔΔΔΔ12.345679
F13.6	123.45678901	ΔΔΔ123.456789
F13.6	-1234.56789012	Δ-1234.567890
F13.6	12345.67890123	Δ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

6.2.7 L Field Descriptor

The L field descriptor specifies the transmission of logical data. It has the form:

Lw

The L field descriptor causes an input statement to read w characters from the external record. If the first non-blank character of that field is the letter T, the value .TRUE. is assigned to the corresponding I/O list element. (The corresponding I/O list element must be of logical type.) If the first non-blank character of the field is the letter F, or if the entire field is blank, the value .FALSE. is assigned. Any other value in the external field causes an error condition.

The L field descriptor causes an output statement to transmit either the letter T, if the value of the corresponding list element is .TRUE., or the letter F, if the value is .FALSE., to an external field w characters wide. The letter T or F is in the rightmost position of the field, preceded by w-1 spaces. For example:

Format	Internal Value	External Representation
L5	.TRUE.	ΔΔΔΔT
L1	.FALSE.	F

6.2.8 A Field Descriptor

The A field descriptor specifies the transmission of alphanumeric data. It has the form:

Aw

On input, the A field descriptor causes w characters to be read from the external record and stored in ASCII format in the corresponding I/O list element. (The corresponding I/O list element may be of any data type.) The maximum number of characters that can be stored in a variable or array element depends on the data type of that element, as follows:

I/O List Element	Maximum Number of Characters	
Logical*1	1	
Logical*2	2	(FORTRAN IV-PLUS only)
Logical*4	4	
Integer*2	2	
Integer*4	4	
Real	4	
Double Precision	8	
Complex	8	

If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost one, two, four, or eight characters (depending on the data type of the variable or array element) are assigned to that entity; the leftmost excess characters are lost. If w is less than the number of characters that can be stored, w characters are assigned to the list

element, left justified, and trailing spaces are added to fill the variable or array element. For example:

Format	External Field	Internal Representation
A6	PAGEΔ#	#Δ (Logical*1)
A6	PAGEΔ#	Δ# (Integer*2)
A6	PAGEΔ#	GEΔ# (Real)
A6	PAGEΔ#	PAGEΔ#ΔΔ (Double Precision)

On output, the A field descriptor causes the contents of the corresponding I/O list element to be transmitted to an external field w characters wide. If the list element contains fewer than w characters, the data appears in the field right-justified with leading spaces. If the list element contains more than w characters, only the leftmost w characters are transmitted. For example:

Format	Internal Value	External Representation
A5	OHMS	ΔOHMS
A5	VOLTSΔΔΔΔ	VOLTS
A5	AMPERESΔ	AMPER

6.2.9 H Field Descriptor

The H field descriptor has the form of a Hollerith constant:

$$nHc_1c_2c_3 \ldots c_n$$

n specifies the number of characters that are to be transmitted

c is an ASCII character.

When the H field descriptor appears in a format specification, data transmission takes place between the external record and the field descriptor itself.

The H field descriptor causes an input statement to read n characters from the external record and to place them in the field descriptor, with the first character appearing immediately after the letter H. Any characters that had been in the field descriptor prior to input are replaced by the input characters.

The H field descriptor causes an output statement to transmit the n characters in the field descriptor following the letter H to the external record. An example of the use of H field descriptors for input and output follows:

```
      TYPE  100
100   FORMAT (41HΔENTERΔPROGRAMΔTITLE,ΔUPΔTOΔ20ΔCHARACTERS)
      ACCEPT  200
200   FORMAT (20HΔΔTITLEΔGOESΔHEREΔΔΔ)
```

The TYPE statement transmits the characters from the H field descriptor in statement 100 to the user's terminal. The ACCEPT statement accepts the response from the keyboard, placing the input data in the H field descriptor in statement 200. The new characters replace the words TITLE GOES HERE; if the user enters fewer than 20 characters, the remainder of the H field descriptor is filled with spaces to the right.

6.2.9.1 Alphanumeric Literals - An alphanumeric literal can be used in place of an H field descriptor. Both types of format specifiers function identically.

The apostrophe character is written within an alphanumeric literal as two apostrophes. For example:

```
50 FORMAT ('TODAY''SΔDATEΔIS:Δ',I2,'/',I2,'/',I2)
```

A pair of apostrophes used in this manner is considered to be a single character.

6.2.10 X Field Descriptor

The X field descriptor has the form:

```
nX
```

The X field descriptor causes an input statement to skip over the next n characters in the input record.

The X field descriptor causes an output statement to transmit n spaces to the external record. For example:

```
      WRITE (5,90) NPAGE
90    FORMAT (13H1PAGEΔNUMBERΔ,I2,16X,23HGRAPHICΔANALYSIS,ΔCONT.)
```

The WRITE statement prints a record similar to:

```
PAGE NUMBER nn                GRAPHIC ANALYSIS, CONT.
```

where "nn" is the current value of the variable NPAGE. The numeral 1 in the first H field descriptor is not printed, but is used to advance the printer paper to the top of a new page. Printer carriage control is explained in Section 6.3.

6.2.11 T Field Descriptor

The T field descriptor is a tabulation specifier. It has the form:

```
Tn
```

where n indicates the character postion of the external record. The value of n must be greater than or equal to one, but not greater than the number of characters allowed in the external record.

On input, the T field descriptor causes the external record to be positioned to its nth character position. For example, if a READ statement inputs a record containing:

```
ABCΔΔΔXYZ
```

under control of the FORMAT statement:

```
10    FORMAT (T7,A3,T1,A3)
```

the READ statement would input the characters XYZ first, then the characters ABC.

On output to devices other than the line printer or terminal, the T field descriptor states that subsequent data transfer is to begin at the nth character position of the external record. For output to a printing device, data transfer begins at position (n-1). The first position of a printed record is reserved for a carriage control character (see Section 6.3) which is never printed. For example the statements:

```
      PRINT 25
25    FORMAT (T51,'COLUMNΔ2',T21,'COLUMNΔ1')
```

would cause the following line to be printed:

```
                    Position 20                          Position 50
                        ↓                                    ↓
                    COLUMN 1                             COLUMN 2
```

6.2.12 Q Field Descriptor

The Q field descriptor, which is simply the letter Q, is used to obtain the number of characters in the input record remaining to be transmitted during a READ operation. The I/O list element corresponding to the Q field descriptor must be of Integer type.

As an example, the statements:

```
      READ (4,1000) XRAY,KK,NCHRS,(ICHR(I),I=1,NCHRS)
1000  FORMAT (E15.7,I4,Q,80A1)
```

reads two fields into the variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS and exactly that many characters are read into the array ICHR. By placing the Q descriptor first in the format specification, the actual length of the input record can be determined.

When the Q field descriptor is used with an output statement it has no effect except that the corresponding list item is skipped.

6.2.13 $ Descriptor

The character $ (dollar sign) appearing in a format specification modifies the carriage control specified by the first character of the record. The $ descriptor is intended primarily for interactive I/O and causes the terminal print position to be left at the end of the text written (rather than returned to the left margin) so that a typed response will appear on the same line following the output.

6.2.14 : Descriptor

The character : (colon) appearing in a format specification causes termination of format control if there are no more items in the I/O list. It has no effect if there are I/O list items remaining. For example the statements:

```
      PRINT   1,3
      PRINT   2,4
1     FORMAT('ΔI=',I2, 'ΔJ=',I2)
2     FORMAT('ΔK=',I2:,'ΔL=',I2)
```

print the following 2 lines:

```
I=Δ3ΔJ=
K=Δ4
```

Section 6.7 describes format control in detail.

6.2.15 Complex Data Editing

Since a complex value is an ordered pair of real values, input or output of a complex entity is governed by two real field descriptors, using any combination of the forms Fw.d, Ew.d, Dw.d or Gw.d.

On input, two successive fields are read and assigned to a complex I/O list element as its real and imaginary parts, respectively. For example

Format	External Fields	Internal Representation
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 12345.678

On output, the constituent parts of a complex value are transmitted under the control of repeated or successive field descriptors. Nothing intervenes between those parts unless explicitly stated by the format specification. For example:

Format	Internal Values	External Representation
2F8.5	2.3547188, 3.456732	Δ2.35472Δ3.45673
E9.2,'Δ,Δ',E5.3	47587.222, 56.123	Δ0.48E+06Δ,Δ*****

6.2.16 Scale Factor

The location of the decimal point in real and double precision values, and in the constituent parts of complex values, can be altered during input or output through the use of a scale factor. The scale factor has the form:

nP

n is a signed or unsigned integer constant in the range -127 to +127 specifying the number of positions the decimal point is to be moved to the right or left.

A scale factor may appear anywhere in a format specification, but must precede the field descriptors with which it is to be associated. It has the forms:

nPFw.d nPEw.d nPDw.d nPGw.d

Data input under control of one of the above field descriptors is multiplied by 10^{-n} before it is assigned to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left; a -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. If the external field contains an explicit exponent, however, the scale factor has no effect. For example:

Format	External Field	Internal Representation
3PE10.5	ΔΔΔ37.614Δ	.037614
3PE10.5	ΔΔ37.614E2	3761.4
-3PE10.5	ΔΔΔΔ37.614	37614.0

The effect of the scale factor on output depends on the type of field descriptor with which it is associated. For the F field descriptor, the value of the I/O list element is multiplied by 10^{n} before being transmitted to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

Values output under control of an E or D field descriptor with scale factor are adjusted by multiplying the basic real constant portion of each value by 10^{n} and subtracting n from the exponent. Thus a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

The effect of the scale factor is suspended while the magnitude of the data to be output is within the effective range of the G field descriptor, since it supplies its own scaling function. The G field descriptor functions as an E field descriptor when the magnitude of the data value is outside its range; the effect of the scale factor is therefore the same as described for that field descriptor.

Note that on input, and on output under control of an F field descriptor, a scale factor actually alters the magnitude of the data; on output, a scale factor attached to an E, D, or G field descriptor merely alters the form in which the data is transmitted. Note also that on input a positive scale factor moves the decimal point to the left and a negative scale factor moves the decimal point to the right, while on output the effect is just the reverse.

If no scale factor is attached to a field descriptor, a scale factor of zero is assumed. Once a scale factor has been specified, however, it applies to all subsequent real and double precision field descriptors in the same format specification, unless another scale factor appears; that scale factor then assumes control. Note that format reversion (Section 6.7) has no effect on the scale factor. A scale factor of zero can only be reinstated by an explicit 0P specification.

Some examples of scale factor effect on output are:

Format	Internal Value	External Representation
1PE12.3	-270.139	ΔΔ-2.701E+02
1PE12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ-0.03E+04

6.2.17 Grouping and Group Repeat Specifications

Any field descriptor (except H, T, P, or X) can be applied to a number of successive data fields by preceding that field descriptor with an unsigned integer constant, called a repeat count, that specifies the number of repetitions. For example, the statements:

```
20   FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
```

and

```
20   FORMAT (3E12.4,4I5)
```

have the same effect.

Similarly, a group of field descriptors can be repeatedly applied to data fields by enclosing those field descriptors in parentheses, with an unsigned integer constant, called a group repeat count, preceding the left parenthesis. For example:

```
50   FORMAT (2I8,3(F8.3,E15.7))
```

is equivalent to:

```
50   FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
                   1         2         3
```

An H or X field descriptor, which could not otherwise be repeated, can be enclosed in parentheses and treated as a group repeat specification, thus allowing it to be repeated a desired number of times.

If a group repeat count is omitted, it is assumed to be 1.

6.2.18 Variable Format Expressions

An expression can be used in a FORMAT statement wherever an integer can be used (except as a Hollerith count) by enclosing it in angle brackets. For example:

```
FORMAT (I<J+1>)
```

causes an I conversion with a field width one greater than the value of J at the time the format is scanned. The expression is re-evaluated each time it is encountered in the normal format scan. If the expression is not of type integer it is converted to integer prior to use. Any valid FORTRAN expression can be used, including function calls and references to dummy arguments.

A complete example is shown in Figure 6-1.

The value of a variable format expression must obey the restrictions on magnitude applying to its use in the format or an error condition results.

```
      DIMENSION A(5)
      DATA A/1.,2.,3.,4.,5./
C
      DO 10 I = 1,10
      WRITE (5,100) I
100   FORMAT(I<MAX0(I,5)>)
10    CONTINUE
C

      DO 20 I = 1,5
      WRITE (5,101) (A(I),J=1,I)
101   FORMAT (<I>F10.<I-1>)
20    CONTINUE
      END
```

produces the following output when executed:

```
1
2
3
4
5
 6
  7
   8
    9
    10
    1.
   2.0       2.0
  3.00      3.00      3.00
 4.000     4.000     4.000     4.000
5.0000    5.0000    5.0000    5.0000    5.0000
```

Figure 6-1
Variable Format Expression Example

6.2.19 Default Field Descriptors

If the field descriptors I, O, L, F, E, D, G, or A are written without any field width value (e.g., F), a default value for w and d is supplied based upon the data type of the I/O list element.

The values for w and d are shown in Table 6-2.

Table 6-2
Default Field Descriptor Values

Field Descriptor	List Element	w	d
I or O	INTEGER*2	7	
I or O	INTEGER*4	12	
L	LOGICAL	2	
F,E,G or D	REAL, COMPLEX	15	7
F,E,G or D	DOUBLE PRECISION	25	16
A	LOGICAL*1	1	
A	LOGICAL*2,INTEGER*2	2	
A	LOGICAL*4,INTEGER*4	4	
A	REAL,COMPLEX	4	
A	DOUBLE PRECISION	8	

Note that for A format, the actual length of the corresponding I/O list element is used.

6.3 CARRIAGE CONTROL

The first character of every record transmitted to a printing device is never printed; instead, it is interpreted as a carriage control character. The FORTRAN I/O system recognizes certain characters for this purpose; the effects of these characters are shown in Table 6-3.

Table 6-3
Carriage Control Characters

Character	Effect
Δ space	Advances one line
0 zero	Advances two lines
1 one	Advances to top of next page
+ plus	Does not advance (allows overprinting)
$ dollar sign	Advances one line before printing and suppresses carriage return at the end of the record

Any character other than those described in Table 6-3 is treated as though it is a space, and is deleted from the print line.

6.4 FORMAT SPECIFICATION SEPARATORS

Field descriptors in a format specification are generally separated from one another by commas. The slash (/) record terminator can also be used to separate field descriptors. A slash causes the input or output of the current record to be terminated and a new record to be initiated. For example:

```
      WRITE (5,40) K,L,M,N,O,P
40    FORMAT (3O6/I6,2F8.4)
```

is equivalent to:

```
      WRITE (5,40) K,L,M
40    FORMAT (3O6)
      WRITE (5,50) N,O,P
50    FORMAT (I6,2F8.4)
```

It is possible to bypass input records or to output blank records by the use of multiple slashes. If n consecutive slashes appear between

two field descriptors, they cause (n-1) records to be skipped on input or (n-1) blank records to be output. (The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.) If n slashes appear at the beginning or end of a format specification, however, they result in n skipped or blank records, because the initial and terminal parentheses of the format specification are themselves a record initiator and record terminator, respectively. An example of the use of multiple record terminators is as follows:

```
      WRITE (5,99)
99    FORMAT ('1'T51'HEADINGΔLINE'//T51'SUBHEADINGΔLINE'//)
```

The above statements output the following:

```
              Column 50, top of page
                         ↓
                         HEADING LINE
               (blank line)
                         SUBHEADING LINE
               (blank line)
               (blank line)
```

6.5 EXTERNAL FIELD SEPARATORS

A field descriptor such as Fw.d specifies that an input statement is to read w characters from the external record. If the data field in question contains fewer than w characters, the input statement would read some characters from the following field unless the short field were padded with leading zeros or spaces. To avoid the necessity of doing so, an input field containing fewer than w characters may be terminated by a comma, which overrides the field descriptor's field width specification. This practice, called short field termination, is particularly useful when entering data from a terminal keyboard. It may be used in conjunction with I, O, F, E, D, G, and L field descriptors. For example:

```
      READ (6,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

If the external record input by the above statements contains:

```
      1,-2,1.0,35
```

Then the following assignments take place:

```
      I = 1

      J = -2

      A = 1.0

      B = 0.35
```

Note that the physical end of the record also serves as a field terminator. Note also that the d part of a w.d specification is not affected as illustrated by the assignment to B.

Only fields of fewer than w characters can be terminated by a comma. If a field of w characters or greater is followed by a comma, the comma will be considered to be part of the following field.

Two successive commas, or a comma following a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor in question, the resulting value assigned is 0, 0.0, 0D0, or .FALSE..

A comma cannot be used to terminate a field that is to be read under control of an A, H, or alphanumeric literal field descriptor. If the physical end of the record is encountered before w characters have been read, however, short field termination is accomplished and the characters that were input are assigned successfully. Trailing spaces are appended to fill the corresponding I/O list element or the field descriptor.

6.6 OBJECT TIME FORMAT

Format specifications may be stored in arrays. Such a format specification (termed an object time format) can be constructed or altered during program execution. The form of a format specification in an array is identical to a FORMAT statement, except that the word FORMAT and the statement label are not present. The initial and terminal parentheses must appear, however. Object-time formats may not be stored in VIRTUAL arrays. An example of object-time format is:

```
      REAL TABLE(5,5)
      DOUBLE PRECISION FORRAY(20),RPAR,FBIG,FMED,FSML
      DATA FORRAY(1),RPAR/'(',')'/
      DATA FBIG,FMED,FSML/'F8.2,','F9.4,','F9.6,'/
      DO 20 J=1,5
          DO 18 I=1,5
              FORRAY(I+1) = FMED
              IF (TABLE(I,J) .GE. 100) FORRAY(I+1) = FBIG
              IF (TABLE(I,J) .LE. 0.1) FORRAY(I+1) = FSML
18        CONTINUE
          FORRAY(I+1) = RPAR
          WRITE (5,FORRAY) (TABLE(K,J),K=1,5)
20    CONTINUE
      END
```

In this example, the DATA statement assigns a left parenthesis to the first element of FORRAY and assigns a right parenthesis and three field descriptors to variables for later use. The proper field descriptors are then selected for inclusion in the format specification, based on the magnitude of the individual elements of array TABLE. A right parenthesis is then added to the format specification just before its use by the WRITE statement. Thus, the format specification changes with each iteration of the DO loop.

6.7 FORMAT CONTROL INTERACTION WITH INPUT/OUTPUT LISTS

Format control is initiated with the beginning of execution of a formatted I/O statement. Each action of format control depends on information provided jointly by the next element of the I/O list (if

one exists) and the next field descriptor of the FORMAT statement or format array. Both the I/O list and the format specification, except for the effects of repeat counts, are interpreted from left to right.

If the I/O statement contains an I/O list, at least one field descriptor of a type other than H, X, T or P must exist in the format specification. An execution error occurs if this condition is not met.

When a formatted input statement is executed, it reads one record from the specified unit and initiates format control; thereafter, additional records can be read as indicated by the format specification. Format control demands that a new record be input whenever a slash is encountered in the format specification, or when the last outer right parenthesis of the format specification is reached and I/O list elements remain to be filled. Any remaining characters in the current record are discarded at the time the new record is read.

When a formatted output statement is executed, it transmits a record to the specified unit as format control terminates. Records may also be output during format control if a slash appears in the format specification or if the last outer right parenthesis is reached and more I/O list elements remain to be transmitted.

Each field descriptor of types I, O, F, E, D, G, L, A, and Q corresponds to one element in the I/O list. No list element corresponds to an H, X, P, T, or alphanumeric literal field descriptor. In the case of H and alphanumeric literal field descriptors, data transfer takes place directly between the external record and the format specification.

When format control encounters an I, O, F, E, D, G, L, A, or Q field descriptor, it determines if a corresponding element exists in the I/O list. If so, format control transmits data, appropriately converted to or from external format, between the record and the list element, then proceeds to the next field descriptor (unless the current one is to be repeated). If there is no corresponding list element, format control terminates.

When the last outer right parenthesis of the format specification is reached, format control determines whether or not there are more I/O list elements to be processed. If not, format control terminates. If additional list elements remain, however, the current record is terminated, a new one initiated, and format control reverts to the rightmost top-level group repeat specification (the one whose left parenthesis matches the next-to-last right parenthesis of the format specification). (This concept is known as format reversion.) If no group repeat specification exists in the format specification, format control returns to the initial left parenthesis of the format specification. Format control continues from that point.

6.8 SUMMARY OF RULES FOR FORMAT STATEMENTS

The following is a summary of the rules pertaining to the construction and use of the FORMAT statement or format array and its components, and to the construction of the external fields and records with which a format specification communicates.

6.8.1 General

1. A FORMAT statement must always be labeled.

2. In a field descriptor such as rIw or nX, the terms r, w, and n must be unsigned integer constants greater than zero. The repeat count and field width specification may be omitted.

3. In a field descriptor such as Fw.d, the term d must be an unsigned integer constant. It must be present in F, E, D, and G field descriptors even if it is zero. The decimal point must also be present. The field width specification, w, must be greater than or equal to d. The w and d must either both be present or both omitted.

4. In a field descriptor such as $nHc_1c_2 \ldots c_n$, exactly n characters must be present following the H format code. Any ASCII character may appear in this field descriptor (an alphanumeric literal field descriptor follows the same rule).

5. In a scale factor of the form nP, n must be a signed or unsigned integer constant in the range -127 to 127 inclusive. Use of the scale factor applies to F, E, D, and G field descriptors only. Once a scale factor has been specified, it applies to all subsequent real or double precision field descriptors in that format specification until another scale factor appears; an explicit 0P specification is required to reinstate a scale factor of zero. Note that format reversion has no effect on the scale factor.

6. No repeat count is permitted in H, X, T or alphanumeric literal descriptors unless those field descriptors are enclosed in parentheses and treated as a group repeat specification.

7. If an I/O list is present in the associated I/O statement, the format specification must contain at least one field descriptor of a type other than H, X, P, T or alphanumeric literal.

8. A format specification in an array must be constructed identically to a format specification in a FORMAT statement, including the initial and terminal parentheses. When a format array name is used in place of a FORMAT statement label in an I/O statement, that name must not be subscripted.

6.8.2 Input

1. An external input field with a negative value must be preceded by a minus symbol; a positive value may optionally be preceded by a plus sign.

2. An external field whose input conversion is governed by an I field descriptor must have the form of an integer constant. An external field input under control of an O field descriptor must have the form of an octal constant (Section 2.4.5), without a leading double quote. Neither can contain a decimal point or an exponent.

3. An external field whose input conversion is handled by an F, E, or G field descriptor must have the form of an integer constant or a real or double precision constant (Section 2.4.2). It can contain a decimal point and/or an E or D exponent field.

4. If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the corresponding real or double precision field descriptor.

5. If an external field contains an exponent, it causes the scale factor (if any) of the corresponding field descriptor to be inoperative for the conversion of that field.

6. The field width specification must be large enough to accommodate, in addition to the numeric character string of the external field, any other characters that can be present (algebraic sign, decimal point, and/or exponent).

7. A comma is the only character that is acceptable for use as an external field separator. It is used to terminate input of fields that are shorter than the number of characters expected, or to designate null (zero-length) fields.

6.8.3 Output

1. A format specification must not demand the output of more characters than can be contained in the external record (for example, a line printer record cannot contain more than 133 characters including the carriage control character).

2. The field width specification, w, must be large enough to accommodate all characters that may be generated by the output conversion, including an algebraic sign, decimal point, and exponent (the field width specification in an E field descriptor, for example, should be large enough to contain (d+7) characters).

3. The first character of a record output to a line printer or terminal is used for carriage control; it is never printed. The first character of such a record should be a space, 0,1, $, or +. Any other character is treated as a space and is deleted from the record.

CHAPTER 7

SPECIFICATION STATEMENTS

This chapter discusses the FORTRAN specification statements. Specification statements are nonexecutable. They provide the information necessary for the proper allocation and initialization of variables and arrays, and define other characteristics of the symbolic names used in the program.

IMPLICIT

7.1 IMPLICIT STATEMENT

The IMPLICIT statement overrides the implied data type of symbolic names, in which all names that begin with the letters I, J, K, L, M, or N are presumed to represent integer data and all names beginning with any other letter are presumed to be real.

The IMPLICIT statement has the form:

```
IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]...
```

typ is one of the following data type specifiers:

```
INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
BYTE
LOGICAL
LOGICAL*1
LOGICAL*4
```

LOGICAL*2

a is an alphabetic specification in either of the general form

c

or

c_1-c_2

c is an alphabetic character.

The latter form specifies a range of letters, from c1 through c2, which must occur in alphabetical order.

The IMPLICIT statement assigns the specified data type to all symbolic names that begin with any specified letter, or any letter within a specified range, and which have no explicit type declaration. For example, the statements:

```
IMPLICIT INTEGER (I,J,K,L,M,N)
IMPLICIT REAL (A-H, O-Z)
```

represent the default in the absence of any data type specifications.

IMPLICIT statements must not be labeled.

Examples

```
IMPLICIT DOUBLE PRECISION D
IMPLICIT COMPLEX (S,Y), LOGICAL*1 (L,A-C)
```

TYPE DEC-

LARATION

7.2 TYPE DECLARATION STATEMENTS

Type declaration statements explicitly define the data type of specified symbolic names.

Type declaration statements have the form:

```
typ v[,v]...
```

typ is one of the following data type specifiers:

```
BYTE
LOGICAL
LOGICAL*1
LOGICAL*4
INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
LOGICAL*2
```

v is the symbolic name of a variable, array, statement function or FUNCTION subprogram, or an array declarator.

A type declaration statement causes the specified symbolic names to have the specified data type; it overrides the data type implied by the initial letter of a symbolic name.

A type declaration statement can define arrays by including array declarators (see Section 2.6) in the list. In each program unit, an array name can appear only once in an array declarator.

A symbolic name can be followed by a data type length specifier of the form *s, where s is one of the acceptable lengths for the data type being declared. Such a specification overrides, for the item with which it was specified, the length attribute implied by the statement. For example:

```
INTEGER*2 I, J, K, M12*4, Q, IVEC*4(10)
REAL*8 WX1, WX2, WX3*4, WX5, WX6*8
```

Type declaration statements should precede all executable statements and all specification statements except the IMPLICIT statement. It must precede the first use of any symbolic name it defines.

The data type of a symbolic name can be explicitly declared only once.

Type declaration statements must not be labeled.

Examples

```
INTEGER COUNT, MATRIX(4,4), SUM
REAL MAN,IABS
LOGICAL SWITCH
```

DIMENSION

7.3 DIMENSION STATEMENT

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

The DIMENSION statement has the form:

```
DIMENSION a(d)[,a(d)]...
```

a is the symbolic name of an array.

d is a dimension declarator.

Each a(d) is an array declarator as described in Section 2.6.

The DIMENSION statement allocates a number of storage locations, one for each element in each dimension, to each array named in the statement. Each storage location is one, two, four or eight bytes in length, as determined by the data type of the array. The total number of locations assigned to an array is equal to the product of all dimension declarators in the array declarator for that array. For example:

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

defines ARRAY as having 16 real elements of two words each, and MATRIX as having 125 integer elements of one word each.

For further information concerning arrays and the storage of array elements, see Section 2.6.

Array declarators can also appear in type declaration, VIRTUAL, and COMMON statements; however, in each program unit, an array name can appear in only one array declarator.

DIMENSION statements must not be labeled.

Examples

```
DIMENSION BUD(12,24,10)

DIMENSION X(5,5,5),Y(4,85),Z(100)

DIMENSION MARK(4,4,4,4)
```

COMMON

7.4 COMMON STATEMENT

A COMMON statement defines one or more contiguous areas (blocks) of storage. Each block is identified by a symbolic name; in addition, one common block is also called the blank common block. A COMMON statement also defines the order of variables and arrays that are part of each common block.

Data in COMMON can be referenced from different program units by the same block name.

The COMMON statement has the form:

```
COMMON [/[cb]/] nlist[[,]/[cb]/ nlist]...
```

cb is a symbolic name, called a common block name, or is blank. If the first cb is blank, the first pair of slashes can be omitted.

nlist is a list of variable names, array names, and array declarators separated by commas.

A common block name can be the same as a variable or array name; however, it cannot be the same as the name of a function or a subroutine, or a function or subroutine entry, in the executable program.

Common blocks with the same name that are declared in different program units all share the same storage area when those program units are combined into an executable program.

Because assignment of components to common is on a one-for-one storage basis, components assigned by a COMMON statement in one program unit should agree in data type with those placed in common by another program unit. For example, if one program unit contains the statement:

```
COMMON CENTS
```

and another program unit contains the statement:

```
COMMON MONEY
```

incorrect results can occur since the 1-word integer variable MONEY is made to correspond to the high-order word of the real variable CENTS.

Care must be taken when LOGICAL*1 elements are assigned to common, to ensure that any data of other types, assigned following the LOGICAL*1 data, is allocated on a word boundary. All common blocks start on a word (even) boundary.

Example

Main Program	Subprogram
COMMON HEAT,X/BLK1/KILO,Q	SUBROUTINE FIGURE
.	COMMON /BLK1/LIMA,R/ /ALFA,BET
.	.
CALL FIGURE	.
.	.
.	RETURN
.	END

The COMMON statement in the main program places HEAT and X in blank common and places KILO and Q in a labeled common block, BLK1. The COMMON statement in the subroutine causes ALFA and BET to correspond to HEAT and X in blank common and makes LIMA and R correspond to KILO and Q in BLK1.

A COMMON statement must not contain the names of VIRTUAL arrays or VIRTUAL array declarators.

7.4.1 Blank Common and Named Common

There can be only one blank common block in an entire executable program. COMMON statements can be used to establish any number of named common blocks.

7.4.2 COMMON Statements with Array Declarators

Array declarators can be used in the COMMON statement to define arrays. Array names must not be otherwise subscripted (individual array elements cannot be assigned to common). In a program unit, an array name can appear only once in an array declarator.

7.5 EQUIVALENCE STATEMENT

EQUIVALENCE

The EQUIVALENCE statement declares two or more entities to be associated (either totally or partially) with the same storage location. The EQUIVALENCE statement references components that exist in the same program unit.

The EQUIVALENCE statement has the form:

```
EQUIVALENCE (nlist) [,(nlist)]...
```

nlist is a list of variables and array elements, separated by commas. At least two components must be present in each nlist.

The EQUIVALENCE statement causes all of the variables or array elements in one parenthesized list to be allocated beginning in the same storage location. Note that an Integer variable made equivalent to a Real variable shares storage with the high-order word of that variable. Mixing of data types in this way is permissible. Multiple components of one data type can share the storage of a single component of a higher-ranked data type. For example:

```
DOUBLE PRECISION DVAR
INTEGER*2  IARR(4)
EQUIVALENCE (DVAR,IARR(1))
```

The EQUIVALENCE statement causes the four elements of the integer array IARR to occupy the same storage as the double precision variable DVAR.

The EQUIVALENCE statement can also be used to equate variable names. For example, the statement

```
EQUIVALENCE (FLTLEN, FLENTH, FLIGHT)
```

causes FLTLEN, FLENTH and FLIGHT to have the same definition provided they are also of the same data type.

An EQUIVALENCE statement in a subprogram must not contain dummy arguments. An EQUIVALENCE statement must not contain the names of VIRTUAL arrays or VIRTUAL array elements.

Examples

```
EQUIVALENCE (A,B), (B,C)          (has the same effect as
                                  EQUIVALENCE (A,B,C))

EQUIVALENCE (A(1),X), (A(2),Y), (A(3),Z)
```

7.5.1 Making Arrays Equivalent

When an element of an array is made equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the corresponding elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both entire arrays are made to share the same storage space. If the third element of a 5-element array is made equivalent to the first element of another array, the last three elements of the first array overlap the first three elements of the second array.

The EQUIVALENCE statement must not attempt to assign the same storage location to two or more elements of the same array, nor to assign memory locations in any way that is inconsistent with the normal linear storage of array elements (for example, making the first element of an array equivalent with the first element of another array, then attempting to set an equivalence between the second element of the first array and the sixth element of the other).

In the EQUIVALENCE statement only, it is possible to identify an array element with a single subscript (i.e., the linear element number), even though the array has been defined as a multi-dimensional array.

For example, the statements:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(4), TRIPLE(7))
```

result in the entire array TABLE sharing a portion of the storage space allocated to array TRIPLE as illustrated in Figure 7-1.

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Figure 7-1
Equivalence of Array Storage

Figure 7-1 also illustrates that the statements

```
EQUIVALENCE (TABLE(1),TRIPLE(4))
```

and

```
EQUIVALENCE (TRIPLE(1,2,2), TABLE(4))
```

result in the same alignment of the two arrays.

Equivalencing arrays with non-unity lower bounds works in a similar fashion. For example, an array defined as A(2:3,4) is a sequence of 8 values. A reference to A(2,2) refers to the third element in the sequence. If array A(2:3,4) is to share storage with array B(2:4,4) the following statement can be used.

```
EQUIVALENCE (A(8), B(10))
```

The entire array A occupies a portion of the storage space allocated to array B as illustrated in Figure 7-2. Note also that EQUIVALENCE (A, B(3)) and EQUIVALENCE (B(10), A(3,4)) would have caused the same alignment of arrays.

Array B		Array A	
Array Element	Element Number	Array Element	Element Number
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

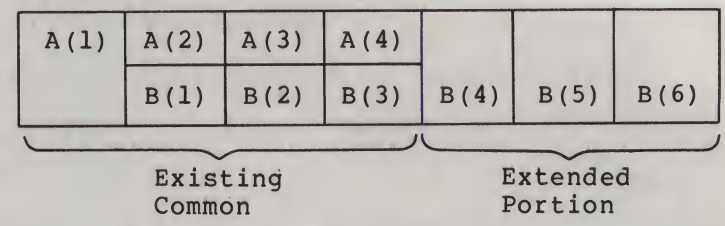
Figure 7-2
Equivalence of Arrays with Non-Unity Lower Bounds

7.5.2 EQUIVALENCE and COMMON Interaction

When components are made equivalent to entities stored in common, the common block can be extended beyond its original boundaries. An EQUIVALENCE statement can only extend common beyond the last element of the previously established common block. It must not attempt to increase the size of common in such a way as to place the extended portion before the first element of existing common. For example:

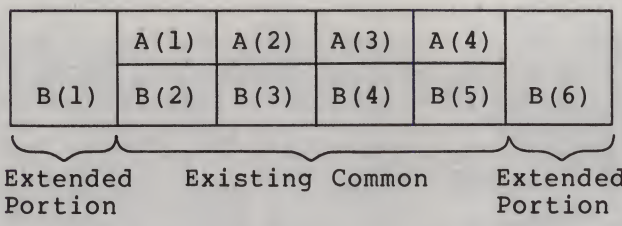
Valid Extension of Common

```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(1))
```



Illegal Extension of Common

```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(3))
```



If two components are assigned to the same or different common blocks, they must not be made equivalent to each other.

7.5.3 EQUIVALENCE and LOGICAL*1 Arrays

If an element of a LOGICAL*1 array that is not aligned on a word boundary is equivalenced to an array or variable of another data type, it can cause that variable or all elements of that array not to be aligned on word boundaries. If this occurs, an attempt to reference that variable or those array elements causes an error during execution of the program.

EXTERNAL

7.6 EXTERNAL STATEMENT

The EXTERNAL statement permits the use of external procedure names (functions, subroutines, and FORTRAN Library Functions) as actual arguments to other subprograms.

The EXTERNAL statement has the form:

```
EXTERNAL v[,v]...
```

v is the symbolic name of a subprogram or the name of a dummy argument which is associated with a subprogram name.

The EXTERNAL statement declares each name in the list to be the name of an external procedure. Such a name can then appear as an actual argument to a subprogram. The subprogram can then use the associated dummy argument name in a function reference or a CALL statement.

Note, however, that a complete function reference used as an argument (such as CALL SUBR(A,SQRT(B),C), for example) represents a value, not a subprogram name; the function name need not be defined in an EXTERNAL statement.

Example

Main Program

```
EXTERNAL SIN,COS,TAN
   .
   .
CALL TRIG (ANGLE,SIN,SINE)
   .
   .
CALL TRIG (ANGLE,COS,COSINE)
   .
   .
CALL TRIG (ANGLE,TAN,TANGNT)
   .
   .
```

Subprograms

```
SUBROUTINE TRIG (X,F,Y)
Y = F(X)
RETURN
END


FUNCTION TAN (X)
TAN = SIN(X) / COS(X)
RETURN
END
```

The CALL statements pass the name of a function to the subroutine TRIG. The function is subsequently invoked by the function reference F(X) in the second statement of TRIG. Thus, the second statement becomes in effect:

```
Y = SIN(X),
Y = COS(X), or
Y = TAN(X)
```

depending upon which CALL statement invoked TRIG (the functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN Library.)

In FORTRAN IV-PLUS, an asterisk (*) can precede a name in the EXTERNAL statement list to designate that the name is that of a user supplied FUNCTION or SUBROUTINE subprogram and is not the name of a FORTRAN library function. See Section 8.2 on FORTRAN Library Functions for additional discussion.

```
EXTERNAL [*]v[,[*]v]...
```

v is the symbolic name of a subprogram.

Example:

```
EXTERNAL COS,*SIN,*TANH,ALPHA
```

DATA

7.7 DATA STATEMENT

The DATA initialization statement permits the assignment of initial values to variables and array elements prior to program execution.

The DATA statement has the form:

```
DATA nlist/clist/[[,]nlist/clist/]...
```

nlist is a list of one or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.

clist is a list of constants.

Constants in a clist have the form:

value

or

n * value

n is a nonzero unsigned integer constant that specifies the number of times the same value is to be assigned to successive entities in the associated nlist.

The DATA statement causes the constant values in each clist to be assigned to the entities in the preceding nlist. Values are assigned in a one-to-one manner in the order in which they appear, from left to right.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array. The associated constant list must therefore contain enough values to fill the array. Array elements are filled in the order of subscript progression.

When Hollerith data is assigned to a variable or array element, the number of characters that can be assigned depends on the data type of the component, as shown in Table 2-2. If the number of characters in a Hollerith constant or alphanumeric literal is less than the capacity of the variable or array element, the constant is extended on the right with spaces. If the number of characters in the constant is greater than the maximum number that can be stored, the rightmost excess characters are not used.

When a Radix-50 constant is assigned to a variable or array element, the number of bytes that can be assigned depends on the data type of the component, as shown in Table 2-2. If the number of bytes of the Radix-50 constant is less than the length of the component, ASCII null characters (zero bytes) are appended on the right. If the number of bytes of the constant exceeds the length of the component, the rightmost excess bytes are not used.

The number of constants in a constant list must correspond exactly to the number of entities specified in the preceding name list. The data types of the data elements and their corresponding symbolic names must agree (except in the case of alphanumeric and Radix-50 data).

The DATA statement must not be used to assign initial values to VIRTUAL arrays or to VIRTUAL array elements.

FORTRAN IV-PLUS converts the constant to the type of the variable being initialized.

Example

```
INTEGER A(10),BELL
DATA A,BELL,STARS/10*0,7, '****'/
```

The DATA statement assigns zero to all ten elements of array A, the value 7 to the variable BELL, and four asterisks to the real variable STARS.

7.8 PARAMETER STATEMENT

The PARAMETER statement allows a constant to be given a symbolic name.

The PARAMETER statement has the form:

```
PARAMETER  p=c  [,p=c] ...
```

p is a symbolic name.

c is a constant.

Each symbolic name, p, becomes a constant and is defined as the value of the constant c.

Once a symbolic name is defined as a constant, it can appear in any position that a constant is allowed; the effect is the same as if the constant were written there instead of the name.

The symbolic name of a constant cannot appear as part of another constant except that the symbolic name of a real constant may appear as a real or imaginary part of a complex constant.

A symbolic name in a PARAMETER statement must not be used to identify anything other than its corresponding constant in that program unit. Such a name cannot appear more than once in PARAMETER statements within the same program unit and therefore cannot be used as a constant in a PARAMETER statement.

The symbolic name of a constant assumes the type implied in the form of its corresponding constant. The initial letter of the name has no effect on its type. Symbolic names of constants cannot appear in type declaration statements except in dimension declarator expressions within array declarators.

PROGRAM

7.9 PROGRAM STATEMENT

The PROGRAM statement, if used, assigns a name to a main program unit.

The PROGRAM statement has the form:

```
PROGRAM nam
```

nam is a symbolic name.

The PROGRAM statement must be the first statement in the main program; its use is optional. The symbolic name may not be the same as the name of any entity within the main program and it must not be the same as the name of any subprogram, entry, or common block in the same executable program.

CHAPTER 8

SUBPROGRAMS

FORTRAN subprograms are divided into two general classes: those that are written by the user and those that are supplied by the FORTRAN system. Subprograms are also grouped into the categories of functions, which includes both arithmetic statement functions and FUNCTION subprograms, and subroutines.

8.1 SUBPROGRAM ARGUMENTS

Arguments to subprograms are represented in two ways: dummy arguments and actual arguments. Dummy arguments are used in the subprogram definition to represent the corresponding actual argument. Dummy arguments appear in the FUNCTION statement, SUBROUTINE statement or arithmetic statement function definition as unsubscripted variable names. Actual arguments appear in the function reference or CALL statement that references the subprogram and provide actual values to be used for computation. Actual arguments can be constants, variables, arrays, array elements, expressions or subprogram names.

Actual and dummy arguments become associated at the time control is transferred to the subprogram. Actual and dummy arguments that become associated must agree in data type. A dummy argument declared as an array can only become associated with an array or array element.

If an actual argument is a constant, expression or subprogram name, the subprogram must not alter the value of the corresponding dummy argument.

Dummy arguments must not appear in COMMON, EQUIVALENCE or DATA statements.

8.2 USER-WRITTEN SUBPROGRAMS

Control is transferred to a function by means of a function reference while control is transferred to a subroutine by a CALL statement. A function reference is the name of the function, together with its arguments, appearing in an expression. A function always returns a value to the calling program. Both functions and subroutines may return additional values via assignment to their arguments. A subprogram can reference other subprograms, but it cannot, either directly or indirectly, reference itself.

8.2.1 Arithmetic Statement Function (ASF)

An arithmetic statement function is a computing procedure defined by a single statement, similar in form to an arithmetic assignment statement. The appearance of a reference to the function within the same program unit causes the computation to be performed and the resulting value to be made available to the expression in which the ASF reference appears.

The arithmetic statement function definition has the form:

```
f ([p[,p]...])=e
```

f is the name of the ASF.

p is a dummy argument.

e is an expression.

The expression is an arithmetic expression that defines the computation to be performed by the ASF.

A function reference to an ASF has the form:

```
f ([p[,p]...])
```

where f is the name of the ASF, and each p is an actual argument.

When a reference to an arithmetic statement function appears in an expression, the values of the actual arguments are associated with the dummy arguments in the ASF definition. The expression in the defining statement is then evaluated and the resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of an ASF is determined either implicitly by the initial letter of the name or explicitly in a type declaration statement.

Dummy arguments in an ASF definition serve only to indicate the number, order, and data type of the actual arguments. The same names may be used to represent other entities elsewhere in the program unit. Note that with the exception of data type, declarative information (such as placement in COMMON or declaration as an array) associated with such an entity is not associated with the ASF dummy arguments. The name of the ASF cannot be used to represent any other entity within the same program unit.

The expression in an ASF definition may contain function references. If a reference to another ASF appears in the expression, that function must have been defined previously.

Any reference to an ASF must appear in the same program unit as the definition of that function.

An ASF reference must appear as, or be part of, an expression; it must not be used as the left side of an assignment statement.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments. Values must have been assigned to them before any reference to the arithmetic statement function.

Examples

ASF Definitions

```
VOLUME(RADIUS) = 4.189*RADIUS**3

SINH(X) = (EXP(X)-EXP(-X))*0.5

AVG(A,B,C,3.) = (A+B+C)/3.      (Invalid; constant as dummy
                                 argument not permitted)
```

ASF References

```
AVG(A,B,C) = (A+B+C)/3.         (Definition)

    .

    .

GRADE = AVG(TEST1,TEST2,XLAB)

IF (AVG(P,D,Q).LT.AVG(X,Y,Z)) GO TO 300

FINAL = AVG(TEST3,TEST4,LAB2)  (Invalid; data type of third
                                argument does not agree with
                                dummy argument)
```

8.2.2 FUNCTION Subprogram

A FUNCTION subprogram is a program unit that consists of a FUNCTION statement followed by a series of statements that define a computing procedure. Control is transferred to a FUNCTION subprogram by a function reference and returned to the calling program unit by a RETURN statement.

A FUNCTION subprogram returns a single value to the calling program unit by assigning that value to the function's name. The data type of the value returned is determined by the function's name.

The FUNCTION statement has the form:

```
[typ] FUNCTION nam[*n][([p[,p]...])]
```

typ is a data type specifier.

nam is a name of the function.

*n is a data type length specifier.

p is a dummy argument.

A function reference that transfers control to a FUNCTION subprogram has the form:

```
nam ([p[,p]...])
```

where nam is the symbolic name of the function, and each p is an actual argument.

When control is transferred to a FUNCTION subprogram, the values supplied by the actual arguments (if any) are associated with the dummy arguments (if any) in the FUNCTION statement. The statements in the subprogram are then executed. The name of the function must be assigned a value before a RETURN statement is executed in that function. When control is returned to the calling program unit, the value thus assigned to the function's name is made available to the expression that contains the function reference, and is used to complete the evaluation of that expression.

The type of a function name may be specified implicitly, explicitly in the FUNCTION statement, or explicitly in a type declaration statement. The type of the function name as defined in the FUNCTION subprogram must be the same as the type of the function name in the calling program unit.

The FUNCTION statement must be the first statement of a function subprogram. It must not be labeled.

A FUNCTION subprogram must not contain a SUBROUTINE statement, a BLOCK DATA statement, or a FUNCTION statement other than the initial statement of the subprogram.

Example

```
      FUNCTION ROOT(A)
      X = 1.0
2     EX = EXP(X)
      EMINX = 1./EX
      ROOT = ((EX+EMINX)*.5+COS(X)-A)/((EX - EMINX)*.5-SIN(X))
      IF (ABS(X-ROOT).LT.1E-6) RETURN
      X = ROOT
      GO TO 2
      END
```

The function in this example uses the Newton-Raphson iteration method to obtain the root of the function:

$$F(X) = \cosh(X) + \cos(X) - A = 0$$

where the value of A is passed as an argument. The iteration formula for this root is:

$$Xi+1 = Xi - \left[\frac{\cosh(Xi)+\cos(Xi)-A}{\sinh(Xi)-\sin(Xi)}\right]$$

which is repeatedly calculated until the difference between Xi and Xi+1 is less than 1×10^{-6}. The function makes use of the FORTRAN Library functions EXP, SIN, COS, and ABS.

8.2.3 SUBROUTINE Subprogram

A SUBROUTINE subprogram is a program unit that consists of a SUBROUTINE statement followed by a series of statements that define a computing procedure. Control is transferred to a SUBROUTINE subprogram by a CALL statement and returned to the calling program unit by a RETURN statement.

The SUBROUTINE statement has the form:

```
SUBROUTINE nam [([p[,p]...])]
```

nam is the name of the subroutine.

p is a dummy argument.

The form of the CALL statement is described in Section 4.5.

When control is transferred to the subroutine, the values supplied by the actual arguments (if any) in the CALL statement are associated with the corresponding dummy arguments (if any) in the SUBROUTINE statement. The statements in the subprogram are then executed.

The SUBROUTINE statement must be the first statement of a subroutine; it must not have a statement label.

A SUBROUTINE subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, or a SUBROUTINE statement other than the initial statement of the subprogram.

Example

Main Program

```
      COMMON NFACES,EDGE,VOLUME
      ACCEPT 65, NFACES,EDGE
65    FORMAT(I2,F8.5)
      CALL PLYVOL
      TYPE 66, VOLUME
66    FORMAT ('ΔVOLUME=',F)
      STOP
      END
```

SUBROUTINE Subprogram

```
      SUBROUTINE PLYVOL
      COMMON NFACES,EDGE,VOLUME
      CUBED = EDGE**3
      GOTO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5,6),NFACES
1     VOLUME = CUBED * 0.11785
      RETURN
2     VOLUME = CUBED
      RETURN
3     VOLUME = CUBED * 0.47140
      RETURN
4     VOLUME = CUBED * 7.66312
      RETURN
5     VOLUME = CUBED * 2.18170
      RETURN
6     TYPE 100, NFACES
100   FORMAT(' NO REGULAR POLYHEDRON HAS ',I3,'FACES.'/)
      RETURN
      END
```

The subroutine in this example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses the computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, and to transfer control to the proper procedure for calculating the

is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, and to transfer control to the proper procedure for calculating the volume. If the number of faces of the body is other than 4,6,8,12, or 20, the subroutine displays an error message on the user's terminal.

ENTRY

8.2.4 ENTRY Statement

The ENTRY statement provides multiple entry points within a subprogram. It is not executable and can appear within a function or subroutine program after the FUNCTION or SUBROUTINE statement. Execution of a subprogram referenced by an entry name begins with the first executable statement following the ENTRY statement.

The ENTRY statement has the form:

```
ENTRY nam [([p[,p]...])]
```

nam is the entry name.

p is a dummy argument.

The ENTRY statement cannot appear within a DO loop.

Entry names appearing in ENTRY statements within SUBROUTINE subprograms must be referenced by CALL statements, and entry names appearing in ENTRY statements within FUNCTION subprograms must be referenced as external function references.

A function entry name can appear in a type declaration statement.

An entry name can appear in an EXTERNAL statement and be used as an actual argument; the entry name in an ENTRY statement cannot be a dummy argument.

Entry names cannot appear in executable statements that physically precede the appearance of the entry name in an ENTRY statement.

The order, number, type and names of the dummy arguments in an ENTRY statement can be different from the order, number, type and names of the dummy arguments in the FUNCTION statement, SUBROUTINE statement, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

8.2.4.1 ENTRY in Function Subprograms - All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, definition of any entry name or the name of the function subprogram causes definition of all the associated names that are of the same type and causes all associated names that are of different type to become undefined. The function and entry names are not required to be the same type, but at the execution of a RETURN statement or the implied return at the end of the subprogram, the name used to reference the function subprogram must be defined. Note that

an entry name cannot appear in executable statements that precede the appearance of the entry name in an ENTRY statement.

8.2.4.2 ENTRY and Array Declarator Interaction - A dummy argument is undefined if it is not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any of the variables appearing in the adjustable array declarator are not currently associated with an actual argument or are not in a COMMON block. Note that there is no retention of argument association between one reference of a subprogram and the next reference of that subprogram. Consider the following example:

```
SUBROUTINE S(A,I,J)
DIMENSION A(I)
A(I) = J
ENTRY S1(I,A,K,L)
A(I) = A(I) + 1
END
```

If B is a real array with 10 elements, as in

```
DIMENSION B(10)
```

then the statement

```
CALL S(B,2,3)
```

would set B(2) = 3 and the statement

```
CALL S1(5,B,3,2)
```

would increment B(5) by 1.

A single function routine that provides the hyperbolic functions sinh, cosh, and tanh appears in Figure 8-1.

```
C
C
      REAL FUNCTION TANH(X)
C
C     ASF TO COMPUTE TWICE SINH
C
      TSINH(Y) = EXP(Y) - EXP (-Y)
C
C     ASF TO COMPUTE TWICE COSH
C
      TCOSH(Y) = EXP(Y) + EXP(-Y)
C
C     COMPUTE TANH
C
      TANH = TSINH(X) / TCOSH(X)
      RETURN
C
C     COMPUTE SINH
C
      ENTRY SINH(X)
      SINH = TSINH(X) / 2.0
      RETURN
C
C     COMPUTE COSH
C
      ENTRY COSH(X)
      COSH = TCOSH(X) / 2.0
      RETURN
C
      END
```

Figure 8-1
Multiple Functions in a Single Function Subprogram

BLOCK DATA

8.2.5 BLOCK DATA Subprogram

The BLOCK DATA subprogram is used to assign initial values to entities in labeled common blocks, at the same time establishing and defining those blocks. It consists of a BLOCK DATA statement followed by a series of specification statements.

The BLOCK DATA statement has the form:

```
BLOCK DATA [nam]
```

nam is a symbolic name.

The statements allowed in a BLOCK DATA subprogram are: Type Declaration, IMPLICIT, DIMENSION, COMMON, EQUIVALENCE, and DATA statements.

The specification statements in the BLOCK DATA subprogram establish and define common blocks, assign variables and arrays to those blocks, and assign initial values to those components.

A BLOCK DATA statement must be the first statement of a BLOCK DATA subprogram. It must not be labeled.

A BLOCK DATA subprogram must not contain any executable statements.

If any entity in a labeled common block is initialized in a BLOCK DATA subprogram, a complete set of specification statements to establish the entire block must be present, even though some of the components in the block do not appear in a DATA statement. Initial values can be defined for more than one block by the same subprogram.

Example

```
BLOCK DATA
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U/AREA2/W,X,Y
DATA R/1.0,2*2.0/,T/.FALSE./,U/0.214537D-7/,W/.TRUE./,Y/3.5/
END
```

8.3 FORTRAN LIBRARY FUNCTIONS

The FORTRAN library functions are listed in Appendix B. Function references to FORTRAN library functions are written in the same form as function references to user-defined functions. For example,

```
R = 3.14159 * ABS(X-1)
```

causes the absolute value of X-1 to be calculated, multiplied by the constant 3.14159, and assigned to the variable R.

The data type of each library function and the data type of the actual arguments is specified in Appendix B. Arguments passed to these functions may not be array names or subprogram names.

8.3.1 Processor-Defined Function References

The FORTRAN library function names are called processor-defined function (PDF) names. Note that the processor-defined functions include both the Intrinsic functions and the Basic External Functions defined in Standard FORTRAN and are treated in a uniform manner.

A name appearing in the table of PDF names normally refers to the FORTRAN library function. The name is assumed to refer to a user-defined function if any of the following conditions exists:

1. The name appears in a type declaration statement specifying a type different from the result type shown in the table.
2. The name appears in an EXTERNAL statement preceded by an asterisk.
3. The name is used in a function reference with arguments of a different type than shown in the table.

Processor-defined function references are local to the program unit in which they occur and do not affect or preclude the use of the name for any other purpose in other program units.

Use of a PDF name in an EXTERNAL statement with a preceding asterisk specifies that the name refers to a function or subroutine that will be provided by the user, and the name becomes a global name in the same sense that the names of functions and subroutines are global.

8.3.2 Generic Function References

Generic function names provide a means by which some of the FORTRAN mathematical functions can be called with selection of the actual library routine used based on the type of the argument that occurs in the function reference. For example, if X is a real variable, then SIN(X) will reference the real valued sine function; while if D is a double precision variable then SIN(D) will reference the double precision sine function; it is not necessary to write DSIN(D).

Generic function selection is performed independently for each function reference, so that in the example above, both SIN(X) and SIN(D) can be used in the same program unit.

The function names for which generic name selection will be performed are shown in Table 8-1. Generic function selection may only be used with the argument types shown in the table.

The names shown in Table 8-1 will lose the generic function selection property if used in a program unit in any of the following ways:

1. In a type declaration statement
2. As the name of an arithmetic statement function
3. As a dummy argument name, common block name, variable or array name.

Using a generic name in an EXTERNAL statement does not affect generic function selection for function references. When one of the generic function names is used in an actual argument list as the name of a function to be passed, that use of the name is not subject to generic selection (there is no argument list on which to base a selection) and the name is treated according to the rules for non-generic FORTRAN functions as described in the preceding section.

The use of a generic function name is local to the program unit in which it occurs and does not affect or preclude the use of that name as a user-written subprogram name, common block name, and so on, in other program units.

Table 8-1
Generic Function Name Summary

Symbolic Name	Type of Argument	Type of Result
ABS	Integer Real Double Complex	Integer Real Double Real
AINT, ANINT	Real Double	Real Double
INT, NINT	Real Double	Integer Integer
SNGL	Integer Double	Real Real
DBLE	Integer Real	Double Double
MOD, MAX, MIN, SIGN, and DIM	Integer Real Double	Integer Real Double
EXP, LOG, SIN, COS, and SQRT	Real Double Complex	Real Double Complex
LOG10, TAN, ATAN, ATAN2, ASIN, ACOS, SINH, COSH, and TANH	Real Double	Real Double

8.3.3 Generic and Processor-Defined Function Usage

The example in Figure 8-2 illustrates the use of generic and processor-defined function references. The name SIN is used in four distinct ways in the figure to emphasize the local and global name properties of each type of use within a single executable program:

1. As the name of an arithmetic statement function
2. As a generic function reference
3. As a processor-defined function reference
4. As a user-defined function

```
C
C       COMPARE WAYS OF COMPUTING SINE.
C
        PROGRAM SINES
        PARAMETER PI = 3.14159265358979323 8D0
        REAL*8 X
        COMMON V(3)
C  NOTE 1: DEFINE SIN AS AN ASF
        SIN(X) = COS(PI/2-X)
        DO 10 X = -PI, PI, 2*PI/100
                  CALL COMPUT(X)
C  NOTE 2:  REFERENCE THE ASF SIN
10                WRITE(6,100) X,V, SIN(X)
100     FORMAT (5F10.7)
        END

C
C
        SUBROUTINE COMPUT(Y)
        REAL*8  Y
C  NOTE 3:  MAKE PDF SIN EXTERNAL FOR USE AT NOTE 5.
        EXTERNAL SIN
        COMMON V(3)
C  NOTE 4:  GENERIC REFERENCE TO D.P. SINE
        V(1) = SIN(Y)
C  NOTE 5:  PDF SINE AS ACTUAL ARGUMENT.
        CALL SUB(Y,SIN)
        END


C
C
        SUBROUTINE SUB(A,S)
C  NOTE 6:  DECLARE SIN AS NAME OF USER FUNCTION.
        EXTERNAL *SIN
C  NOTE 7:  DECLARE SIN AS TYPE REAL*8.
        REAL*8 A, SIN
        COMMON V(3)
C  NOTE 8:  EVALUATE PDF SIN PASSED AT NOTE 5.
        V(2) = S(A)
C  NOTE 9:  EVALUATE USER DEFINED SIN FUNCTION.
        V(3) = SIN(A)
        END


C
C
C  NOTE 10:  DEFINE THE USER SIN FUNCTION.
        REAL*8 FUNCTION SIN(X)
        INTEGER FACTOR
        SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)
        1       - X**7/FACTOR(7)
        END


        INTEGER FUNCTION FACTOR(N)
        FACTOR = 1
        DO 10 I=N, 1, -1
10      FACTOR = FACTOR * I
        END
```

Figure 8-2
Multiple Function Name Usage
(Notes are discussed in the text.)

The following comments elaborate on the notes contained in Figure 8-2.

Note	Comment
1.	An arithmetic statement function named SIN is defined in terms of the generic function name COS. Since the argument of COS is double precision, the double precision cosine function will be evaluated.
2.	The arithmetic statement function SIN is called.
3.	The name SIN is declared external so that the single precision processor-defined sine function may be passed as an actual argument at note 5.
4.	The generic function name SIN is used to reference the double precision sine function.
5.	The single precision processor-defined sine function is used as an actual argument.
6.	The name SIN is declared a user-defined function name.
7.	The type of SIN is declared to be double precision.
8.	The single precision sine function passed at note 5 is evaluated.
9.	The user-defined SIN function is evaluated.
10.	The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

APPENDIX A

CHARACTER SETS

A.1 FORTRAN CHARACTER SET

The FORTRAN character set consists of:

1. The letters A through Z and a through z
2. The numerals 0 through 9
3. The following special characters:

Character	Name
Δ	Space or Blank or Tab
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe
"	Double Quote
$	Dollar Sign
:	Colon
<	Left Angle Bracket
>	Right Angle Bracket

Other printable characters may appear in a FORTRAN statement only as part of a Hollerith constant or alphanumeric literal. Any printable character may appear in a comment.

A.2 ASCII CHARACTER SET

Decimal Value	ASCII Character	Usage
Ø	NUL	FILL character
1	SOH	
2	STX	
3	ETX	CTRL/C
4	EOT	
5	ENQ	
6	ACK	
7	BEL	BELL
8	BS	
9	HT	HORIZONTAL TAB
1Ø	LF	LINE FEED
11	VT	VERTICAL TAB
12	FF	FORM FEED
13	CR	CARRIAGE RETURN
14	SO	
15	SI	CTRL/O
16	DLE	
17	DC1	
18	DC2	
19	DC3	
2Ø	DC4	
21	NAK	CTRL/U
22	SYN	
23	ETB	
24	CAN	
25	EM	
26	SUB	CTRL/Z
27	ESC	ESCAPE[1]
28	FS	
29	GS	
3Ø	RS	
31	US	
32	SP	SPACE
33	!	
34	"	
35	#	
36	$	
37	%	
38	&	
39	'	APOSTROPHE
4Ø	(	
41	)	
42	*	
43	+	
44	,	COMMA
45	-	
46	.	
47	/	
48	Ø	
49	1	
5Ø	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
6Ø	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
7Ø	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
8Ø	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
9Ø	Z	
91	[	
92	\	Backslash
93	]	
94	^	or ↑
95	_	or ←
96	`	Grave accent
97	a	
98	b	
99	c	
1ØØ	d	
1Ø1	e	
1Ø2	f	
1Ø3	g	
1Ø4	h	
1Ø5	i	
1Ø6	j	
1Ø7	k	
1Ø8	l	
1Ø9	m	
11Ø	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
12Ø	x	
121	y	
122	z	
123	{	
124	|	Vertical Line
125	}	
126	~	Tilde
127	DEL	RUBOUT

[1]ALTMODE (ASCII 125) or PREFIX (ASCII 126) keys which appear on some terminals are translated internally into ESCAPE.

A.3 RADIX-50 CHARACTER SET

Radix-50 is a special character data representation in which up to three characters from the Radix-50 character set (a subset of the ASCII character set) can be encoded and packed into a single PDP-11 storage word.

The Radix-50 characters and their corresponding code values are as follows:

Character	ASCII Octal Equivalent	Radix-50 Value (Octal)
Space	40	0
A - Z	101 - 132	1 - 32
$	44	33
.	56	34
(Unassigned)		35
0 - 9	60 - 71	36 - 47

Radix-50 values are stored, up to three characters per word, by packing them into single numeric values according to the formula:

```
((i * 50 + j) * 50 + k)
```

where "i", "j", and "k" represent the code values of three Radix-50 characters.

The maximum Radix-50 value is, thus,

```
47*50*50 + 47*50 + 47 = 174777
```

The following table provides a convenient means of translating between the ASCII character set and its Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent is (arithmetic is performed in octal):

```
X   = 113000
 2  = 002400
  B = 000002
-------------
X2B = 115402
```

Radix-50 Character/Position Table

Single Char. or First Char.		Second Character		Third Character	
Space	000000	Space	000000	Space	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
UNUSED	132500	UNUSED	002210	UNUSED	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

APPENDIX B

FORTRAN LANGUAGE SUMMARY

B.1 EXPRESSION OPERATORS

Operators in each type are shown in order of descending precedence.

Type	Operator		Operates Upon
Arithmetic	**	exponentiation	arithmetic or logical constants, variables, and expressions
	*,/	multiplication, division	
	+,-	addition, subtraction unary plus and minus	
Relational	.GT.	greater than	arithmetic or logical constants, variables, and expressions (all relational operators have equal priority)
	.GE.	greater than or equal to	
	.LT.	less than	
	.LE.	less than or equal to	
	.EQ.	equal to	
	.NE.	not equal to	
Logical	.NOT.	.NOT.A is true if and only if A is false	logical or integer constants, variables, and expressions
	.AND.	A.AND.B is true if and only if A and B are both true	
	.OR.	A.OR.B is true if and only if either A or B or both are true	
	.EQV.	A.EQV.B is true if and only if A and B are both true or A and B are both false.	(precedence same as .XOR.)
	.XOR.	A.XOR.B is true if and only if A is true and B is false or B is true and A is false.	(precedence same as .EQV.)

B.2 STATEMENTS

The following summary of statements available in the PDP-11 FORTRAN language defines the general format for the statement. If more detailed information is needed, the reader is referred to the Section(s) in the manual dealing with that particular statement.

Statement Formats	Effect	Manual Section
ACCEPT	See READ, Formatted Sequential See READ, List-Directed	5.4.3 5.7.3

Arithmetic/Logical Assignment

3.1
3.2

v=e

- v — is a variable name or an array element name.
- e — is an expression.

The value of the arithmetic or logical expression is assigned to the variable.

Arithmetic Statement Function

8.2.1

f([p[,p]...])=e

- f — is a symbolic name.
- p — is a symbolic name.
- e — is an expression.

Creates a user-defined function having the variables p as dummy arguments. When referenced, the expression is evaluated using the actual arguments in the function call.

ASSIGN s TO v

3.3

- s — is an executable statement label.
- v — is an integer variable name.

Associate the statement number s with the integer variable v for later use in an assigned GO TO statement.

BACKSPACE u 5.9.2

u is an integer variable or constant.

The currently open file on logical unit u is backspaced one record.

BLOCK DATA [nam] 8.2.5

nam is a symbolic name.

Specifies the subprogram which follows as a BLOCK DATA subprogram.

CALL s[([a][,[a]]...)] 4.5

s is a subprogram name.

a is an expression, a procedure name, or an array name.

Calls the SUBROUTINE subprogram with the name specified by s, passing the actual arguments a to replace the dummy arguments in the SUBROUTINE definition.

CLOSE (p[,p]...) 5.9.7

p is one of the following forms:

```
UNIT = e
DISPOSE = 'SAVE'     or  DISP = 'SAVE'
DISPOSE = 'KEEP'     or  DISP = 'KEEP'
DISPOSE = 'DELETE'   or  DISP = 'DELETE'
DISPOSE = 'PRINT'    or  DISP = 'PRINT'
ERR = s
```

e is a numeric expression
s is an executable statement label

Closes the specified file.

COMMON [/[cb]/] nlist [[,]/[cb]/nlist]... 7.4

cb is a common block name.

nlist is a list of one or more variable names, array names, or array declarators separated by commas.

Reserves one or more blocks of storage space under the name specified to contain the variables associated with that block name.

CONTINUE 4.4

Causes no processing.

DATA nlist/clist/[[,] nlist/clist/]... 7.7

nlist is a list of one or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.

clist is a list of one or more constants separated by commas, each optionally preceded by j*, where j is a nonzero, unsigned integer constant.

Causes elements in the list of values to be initially stored in the corresponding elements of the list of variable names.

DECODE (c,f,b[,ERR=s])[list] 5.10

c is an integer expression.

f is a FORMAT statement label or array name.

b is a variable name, array name, or array element name.

s is an executable statement label.

list is an I/O list.

Changes the elements in the I/O list from character into internal format; c specifies the number of characters, f specifies the format, and b is the name of the entity containing the characters to be converted.

DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]... 5.9.4

u is an integer variable name or integer constant.

m is an integer variable name or integer constant.

n is an integer variable name or integer constant.

v is an integer variable name.

Defines the record structure of a direct access file where u is the logical unit number, m is the number of fixed length records in the file, n is the length in words of a single record, U is a fixed argument, and v is the associated variable.

DIMENSION a(d)[,a(d)]... 7.3

a(d) is an array declarator.

Specifies storage space requirements for arrays.

DO s [,] i = e1,e2[,e3] 4.3

s is the label of an executable statement.

i is a variable name.

ei are integer expressions.

To execute the DO loop:

1. Set i = e1
2. Execute statements through statement number s
3. Evaluate i = i+e3
4. Repeat 2 through 3 for MAX (1, INT((e2 - e1)/e3) + 1) iterations

ENCODE (c,f,b[,ERR=s])[list] 5.10

c is an integer expression.

f is a FORMAT statement label or an array name.

b is a variable name, array name, or array element name.

s is an executable statement label.

list is an I/O list.

Changes the elements in the list of variables into characters; c specifies the number of characters in the buffer, f specifies the format statement number, and b is the name of the entity to be used as a buffer.

END 4.9

Delimits a program unit.

END FILE u 5.9.3

u is an integer variable or constant.

An end-file record is written on logical unit u.

END=s,ERR=s 5.8

s is an executable statement label.

(Transfer of Control) on end-of-file or error condition is an optional element in each type of I/O statement allowing the program to transfer to statement number s on an end-of-file (END=) or error (ERR=) condition.

ENTRY nam [([p[,p]...])] 8.2.4

nam is a subprogram name.

p is a symbolic name.

Defines an alternate entry point within a SUBROUTINE or FUNCTION subprogram.

EQUIVALENCE (nlist)[,(nlist)]... 7.5

nlist is a list of two or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.

Each of the names (nlist) within a set of parentheses is assigned the same storage location.

EXTERNAL v[,v]... 7.6

v is a subprogram name.

Defines the names specified as FUNCTION or SUBROUTINE subprograms.

EXTERNAL [*]v[,[*]v]... 7.6

v is a subprogram name.

Defines the names specified as user-defined subprograms.

FIND (u'r) 5.9.5

u is an integer variable name or integer constant.

r is an integer expression.

Positions the file on logical unit u to record r and sets associated variable to record number r.

FORMAT (field specification,...) 6.1 - 6.8

Describes the format in which one or more records are to be transmitted; a statement label must be present.

[typ] FUNCTION nam[*n][([p[,p]...])] 8.2.2

typ is a data type specifier.

nam is a symbolic name.

*n is a data type length specifier.

p is a symbolic name.

Begins a FUNCTION subprogram, indicating the program name and any dummy argument names, p. An optional type specification can be included.

GO TO s 4.1.1

s is an executable statement label.

(Unconditional GO TO) Transfers control to statement number s.

GO TO (slist)[,] e 4.1.2

slist is a list of one or more executable statement labels separated by commas.

e is an integer expression.

(Computed GO TO) Transfers control to the statement label specified by the value of expression e. (If e=1 control transfers to the first statement label. If e=2 it transfers to the second statement label. etc.) If e is less than 1 or greater than the number of statement labels present, no transfer takes place.

GO TO v [[,](slist)] 4.1.3

v is an integer variable name.

slist is a list of one or more executable statement labels separated by commas.

(Assigned GO TO) Transfers control to the statement most recently associated with v by an ASSIGN statement.

IF (e) s1,s2,s3 4.2.1

e is an expression.

si are executable statement labels.

(Arithmetic IF) Transfers control to statement number si depending upon the value of the expression. If the value of the expression is less than zero, transfer to s1; if the value of the expression is equal to zero, transfer to s2; if the value of the expression is greater than zero, transfer to s3.

IF (e) st 4.2.2

e is an expression.

st is any executable statement except a DO or logical IF statement.

(Logical IF) Executes the statement if the logical expression is true.

IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]... 7.1

typ is a data type specifier.

a is either a single letter, or two letters in alphabetical order separated by a dash (i.e., x-y).

The elements a represent single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that type.

INCLUDE fln 1.5

fln is an alphanumeric literal.

Includes the source statements in the compilation from the file specified by fln.

OPEN (p[,p]...) 5.9.6

p is one of the following forms:

```
UNIT = e
NAME = n
TYPE = 'OLD'
TYPE = 'NEW'
TYPE = 'SCRATCH'
TYPE = 'UNKNOWN'
ACCESS = 'SEQUENTIAL'
ACCESS = 'DIRECT'
ACCESS = 'APPEND'
READONLY
FORM = 'FORMATTED'
FORM = 'UNFORMATTED'
RECORDSIZE = e
ERR = s
BUFFERCOUNT = e
INITIALSIZE = e
EXTENDSIZE = e
NOSPANBLOCKS
SHARED
DISPOSE = 'SAVE'   or DISP = 'SAVE'
DISPOSE = 'KEEP'   or DISP = 'KEEP'
DISPOSE = 'DELETE' or DISP = 'DELETE'
DISPOSE = 'PRINT'  or DISP = 'PRINT'
ASSOCIATEVARIABLE = v
CARRIAGECONTROL = 'FORTRAN'
CARRIAGECONTROL = 'LIST'
CARRIAGECONTROL = 'NONE'
MAXREC = e
BLOCKSIZE = e
```

e is an integer expression.
s is an executable statement label.
v is an integer variable name.
n is an array name, variable name, array element name, or alphanumeric literal.

Opens a file on the specified logical unit according to the parameters specified by the keywords.

PARAMETER p=c [,p=c]... 7.8

p is a symbolic name.

c is a constant.

Defines a symbolic name for a constant.

PAUSE [disp] 4.7

disp is a decimal digit string containing one to five digits, an alphanumeric literal, or an octal constant.

Suspends program execution and prints the display, if one is specified.

PRINT See WRITE, Formatted Sequential 5.4.5
See WRITE, List-Directed 5.7.5

PROGRAM nam 7.9

nam is a symbolic name.

Specifies a name for the main program.

READ (u,f[,END=s][,ERR=s])[list] 5.4.1

READ f[,list]

ACCEPT f[,list]

u is an integer variable or constant.

f is a FORMAT statement label or an array name.

s is an executable statement label.

list is an I/O list.

(Formatted Sequential) Reads one or more logical records from unit u and assigns values to the elements in the list, converted according to format specification f.

READ (u'r,f[,ERR=s])[list] 5.6.1

u is an integer variable or constant.

r is an integer expression.

f is a FORMAT statement label or an array name.

s is an executable statement label.

list is an I/O list.

(Formatted Direct Access) Reads record r from unit u and assigns values to the elements in the list, converted according to format specification f.

```
READ(u[,END=s][,ERR=s])[list]
```

5.3.1

u	is an integer variable or constant.
s	is an executable statement label.
list	is an I/O list.

(Unformatted Sequential) Reads one unformatted record from unit u, and assigns values to the elements in the list.

```
READ(u'r[,ERR=s])[list]
```

5.5.1

u	is an integer variable or constant.
r	is an integer expression.
s	is an executable statement label.
list	is an I/O list.

(Unformatted Direct Access) Reads record r from unit u, and assigns values to the elements in the list.

```
READ (u,*[,END=s][,ERR=s])list
READ *,list
ACCEPT *,list
```

5.7.1

u	is an integer variable or constant.
*	denotes list-directed formatting.
s	is an executable statement label.
list	is an I/O list.

(List-Directed) Reads one or more logical records from unit u and assigns values to the elements in the list, converted according to the data type of the list element.

RETURN 4.6

Returns control to the calling program from the current subprogram.

REWIND u 5.9.1

u is an integer variable or constant.

Repositions logical unit u to the beginning of the currently opened file.

STOP [disp] 4.8

disp is a decimal digit string containing one to five digits, an alphanumeric literal, or an octal constant.

Terminate program execution and print the display, if one is specified.

SUBROUTINE nam[([p[,p]...])] 8.2.3

nam is a symbolic name.

p is a symbolic name.

Begins a SUBROUTINE subprogram, indicating the program name and any dummy argument names, p.

TYPE See WRITE, Formatted Sequential 5.4.4
See WRITE, List-Directed 5.7.4

Type Declaration 7.2

typ v[,v]...

typ is a data type specifier.

v is a variable name, array name, function or function entry name, or an array declarator. The name can optionally be followed by a data type length specifier (*n).

The symbolic names, v, are assigned the specified data type in the program unit.

typ is one of:

DOUBLE PRECISION
COMPLEX
COMPLEX*8
REAL

REAL*4
REAL*8
INTEGER
INTEGER*2
INTEGER*4
BYTE
LOGICAL
LOGICAL*1
LOGICAL*4

LOGICAL*2

VIRTUAL a(d) [,a(d)]... D.1

a(d) is an array declarator that specifies storage space for a VIRTUAL array.

WRITE (u,f[,ERR=s])[list] 5.4.2

PRINT f[,list]

TYPE f[,list]

u is an integer variable or constant.

f is a FORMAT statement label or an array name.

s is an executable statement label.

list is an I/O list.

(Formatted Sequential) Writes one or more logical records to unit u containing the values of the elements in the list, converted according to format specification f.

WRITE (u'r,f[,ERR=s])[list] 5.6.2

u is an integer variable or constant.

r is an integer expression.

f is a FORMAT statement label or an array name.

s is an executable statement label.

list is an I/O list.

(Formatted Direct Access) Writes record r to unit u containing the values of the elements in the list, converted according to format specification f.

WRITE (u[,ERR=s])[list] 5.3.2

u is an integer variable or constant.

s is an executable statement label.

list is an I/O list.

(Unformatted Sequential) Writes one unformatted record to unit u containing the values of the elements in the list.

WRITE (u'r[,ERR=s]) [list] 5.5.2

u is an integer variable or constant.

r is an integer expression.

s is an executable statement label.

list is an I/O list.

(Unformatted Direct Access) Writes record r to unit u containing the values of the elements in the list.

WRITE(u,*[,ERR=s])list 5.7.2

PRINT *,list

TYPE *,list

u is an integer variable or constant.

* denotes list-directed formatting.

s is an executable statement label.

list is an I/O list.

(List-Directed) Writes one or more logical records to unit u containing the values of the elements in the list, converted according to the data type of the list element.

Table B-1
FORTRAN Library Functions

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
ABS(X)	Real absolute value	Real	Real
IABS(I)	Integer absolute value	Integer	Integer
DABS(X)	Double precision absolute value	Double	Double
CABS(Z)	Complex to Real, absolute value where Z=(x,y) CABS(Z)=$(x^2+y^2)^{1/2}$	Complex	Real
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion IFIX(X) is equivalent to INT(X)	Real	Integer
SNGL(X)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double
REAL(Z)	Complex to Real conversion, obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X,Y)	Real to Complex conversion CMPLX(X,Y)=X+i*Y	Real	Complex
	Truncation functions return the sign of the argument * largest integer ≤ |arg|		
AINT(X)	Real to Real truncation	Real	Real
INT(X)	Real to Integer truncation	Real	Integer
IDINT(X)	Double to Integer truncation	Double	Integer
	Remainder functions return the remainder when the first argument is divided by the second.		
AMOD(X,Y)	Real remainder	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder	Double	Double
	Maximum value functions return the largest value from among the argument list; ≥ 2 arguments.		
AMAXØ(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAXØ(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real list	Real	Integer
DMAX1(X,Y,...)	Double maximum from Double list	Double	Double
	Minimum value functions return the smallest value from among the argument list; ≥ 2 arguments.		
AMINØ(I,J,...)	Real minimum of Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum of Real list	Real	Real
MINØ(I,J,...)	Integer minimum of Integer list	Integer	Integer
MIN1(X,Y,...)	Integer minimum of Real list	Real	Integer
DMIN1(X,Y,...)	Double minimum of Double list	Double	Double

Table B-1 (Cont.)
FORTRAN Library Functions

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
	The transfer of sign functions return (sign of the second argument) * (absolute value of first argument).		
SIGN(X,Y)	Real transfer of sign	Real	Real
ISIGN(I,J)	Integer transfer of sign	Integer	Integer
DSIGN(X,Y)	Double precision transfer of sign	Double	Double
	Positive difference functions return the first argument minus the minimum of the two arguments.		
DIM(X,Y)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer
	Exponential functions return the value of e raised to the argument power.		
EXP(X)	e^x	Real	Real
DEXP(X)	e^x	Double	Double
CEXP(Z)	e^z	Complex	Complex
ALOG(X)	Returns $\log_e(X)$	Real	Real
ALOG10(X)	Returns $\log_{10}(X)$	Real	Real
DLOG(X)	Returns $\log_e(X)$	Double	Double
DLOG10(X)	Returns $\log_{10}(X)$	Double	Double
CLOG(Z)	Returns $\log_e$ of complex argument	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z)	Complex cosine	Complex	Complex
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double	Double
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)	Double precision arc tangent of (X/Y)	Double	Double
CONJG(Z)	Complex conjugate, if Z=X+i*Y CONJG(Z)=X-i*Y	Complex	Complex
RAN(I,J)	Returns a random number of uniform distribution over the range 0 to 1. I and J must be integer variables and should be set initially to 0. Resetting I and J to 0 regenerates the random number sequence. Alternate starting values for I and J will generate different random number sequences. See also Appendix C.3.	Integer	Real

B.3 LIBRARY FUNCTIONS

Table B-2
FORTRAN IV-PLUS
Generic and Processor-Defined Functions

Function	Definition	Number of Arguments	Generic Name	Type of Argument(s)	Type of Result	PDF Name	Internal Name
Absolute Value	$\|a\|$	1	ABS	Real Double Complex Integer-2 Integer-4	Real Double Real Integer-2 Integer-4	ABS DABS CABS IIABS JIABS	$ABS $DABS $CABS $IABS $JABS
			IABS	Integer-2 Integer-4	Integer-2 Integer-4	IIABS JIABS	$IABS $JABS
Cosine	Cos a	1	COS	Real Double Complex	Real Double Complex	COS DCOS CCOS	$COS $DCOS $CCOS
Exponential	e^a	1	EXP	Real Double Complex	Real Double Complex	EXP DEXP CEXP	$EXP $DEXP $CEXP
Natural Logarithm	$\log_e a$	1	LOG	Real Double Complex	Real Double Complex	ALOG DLOG CLOG	$ALOG $DLOG $CLOG
Common Logarithm	$\log_{10} a$	1	LOG1Ø	Real Double	Real Double	ALOG1Ø DLOG1Ø	$ALG1Ø $DLG1Ø
Sine	Sin a	1	SIN	Real Double Complex	Real Double Complex	SIN DSIN CSIN	$SIN $DSIN $CSIN
Square Root	$a^{1/2}$	1	SQRT	Real Double Complex	Real Double Complex	SQRT DSQRT CSQRT	$SQRT $DSQRT $CSQRT
Tangent	Tan a	1	TAN	Real Double	Real Double	TAN DTAN	$TAN $DTAN
Arc Cosine	Arc Cos a	1	ACOS	Real Double	Real Double	ACOS DACOS	$ACOS $DACOS
Arc Sine	Arc Sin a	1	ASIN	Real Double	Real Double	ASIN DASIN	$ASIN $DASIN
Arc Tangent	Arc Tan a	1	ATAN	Real Double	Real Double	ATAN DATAN	$ATAN $DATAN
	Arc Tan a_1/a_2	2	ATAN2	Real Double	Real Double	ATAN2 DATAN2	$ATAN2 $DATN2
Hyperbolic Cosine	Cosh a	1	COSH	Real Double	Real Double	COSH DCOSH	$COSH $DCOSH
Hyperbolic Sine	Sinh a	1	SINH	Real Double	Real Double	SINH DSINH	$SINH $DSINH
Hyperbolic Tangent	Tanh a	1	TANH	Real Double	Real Double	TANH DTANH	$TANH $DTANH

Table B-2 (Cont.)
FORTRAN IV-PLUS
Generic and Processor-Defined Functions

Function	Definition	Number of Arguments	Generic Name	Type of Argument(s)	Type of Result	PDF Name	Internal Name
Maximum	$max(a_1,a_2...a_n)$	n	MAX	Integer-2 Integer-4 Real Double	Integer-2 Integer-4 Real Double	IMAXØ JMAXØ AMAX1 DMAX1	$MAXØ $JMAXØ $AMAX1 $DMAX1
			MAXØ	Integer-2 Integer-4	Integer-2 Integer-4	IMAXØ JMAXØ	$MAXØ $JMAXØ
			MAX1*	Real Real	Integer-2 Integer-4	IMAX1 JMAX1	$MAX1 $JMAX1
			AMAXØ	Integer-2 Integer-4	Real Real	AIMAXØ AJMAXØ	$AMAXØ $AJMXØ
Minimum	$min(a_1,a_2...a_n)$	n	MIN	Integer-2 Integer-4 Real Double	Integer-2 Integer-4 Real Double	IMINØ JMINØ AMIN1 DMIN1	$MINØ $JMINØ $AMIN1 $DMIN1
			MINØ	Integer-2 Integer-4	Integer-2 Integer-4	IMINØ JMINØ	$MINØ $JMINØ
			MIN1*	Real Real	Integer-2 Integer-4	IMIN1 JMIN1	$MIN1 $JMIN1
			AMINØ	Integer-2 Integer-4	Real Real	AIMINØ AJMINØ	$AMINØ $AJMNØ
Truncation	[a]	1	INT*	Real Real Double Double	Integer-2 Integer-4 Integer-2 Integer-4	IINT JINT IIDINT JIDINT	$INT $JNT $IDINT $JDINT
			IDINT*	Double Double	Integer-2 Integer-4	IIDINT JIDINT	$IDINT $JDINT
			AINT	Real Double	Real Double	AINT DINT	$AINT $DINT
Nearest Integer	[a+.5*Sign(a)]	1	NINT*	Real Real Double Double	Integer-2 Integer-4 Integer-2 Integer-4	ININT JNINT IIDNNT JIDNNT	$NINT $NJNT $IDNNT $JDNNT
			IDNINT*	Double Double	Integer-2 Integer-4	IIDNNT JIDNNT	$IDNNT $JDNNT
			ANINT	Real Double	Real Double	ANINT DNINT	$ANINT $DNINT

* = Result Generic

Table B-2 (Cont.)
FORTRAN-IV PLUS
Generic and Processor-Defined Functions

Function	Definition	Number of Arguments	Generic Name	Type of Argument(s)	Type of Result	PDF Name	Internal Name
Positive Difference	$a_1-(\min(a_1,a_2))$	2	DIM	Integer-2 Integer-4 Real Double	Integer-2 Integer-4 Real Double	IIDIM JIDIM DIM DDIM	$IDIM $JDIM $DIM $DDIM
			IDIM	Integer-2 Integer-4	Integer-2 Integer-4	IIDIM JIDIM	$IDIM $JDIM
Float	Integer to Real Conversion	1	FLOAT	Integer-2 Integer-4	Real Real	FLOATI FLOATJ	$FLOAT $FLOTJ
Double Precision Float	Integer to Double Conversion	1	DFLOAT	Integer-2 Integer-4	Double Double	DFLOTI DFLOTJ	$DFLTI $DFLTJ
Fix	Real to Integer Conversion	1	IFIX*	Real Real	Integer-2 Integer-4	IIFIX JIFIX	$IFIX $JFIX
Conversion to Single		1	SNGL	Double Real Integer-2 Integer-4	Real Real Real Real	SNGL - FLOATI FLOATJ	$SNGL - $FLOAT $FLOTJ
Conversion to Double		1	DBLE	Real Double Integer-2 Integer-4	Double Double Double Double	DBLE - DFLOTI DFLOTJ	$DBLE - $DFLTI $DFLTJ
Real Part of Complex		1		Complex	Real	REAL	$REAL
Imaginary Part of Complex		1		Complex	Real	AIMAG	$AIMAG
Complex From two Reals		2		Real	Complex	CMPLX	$CMPLX
Complex Conjugate		1		Complex	Complex	CONJG	$CONJG
Double Product of Reals		2		Real	Double	DPROD	$DPROD
Remainder	Remainder when a_1 divided by a_2	2	MOD	Integer-2 Integer-4 Real Double	Integer-2 Integer-4 Real Double	IMOD JMOD AMOD DMOD	$IMOD $JMOD $AMOD $DMOD
Transfer of Sign	Sign $a_2*\lvert a_1\rvert$	2	SIGN	Integer-2 Integer-4 Real Double	Integer-2 Integer-4 Real Double	IISIGN JISIGN SIGN DSIGN	$ISIGN $JSIGN $SIGN $DSIGN
			ISIGN	Integer-2 Integer-4	Integer-2 Integer-4	IISIGN JSIGN	$ISIGN $JSIGN

* = Result Generic

Table B-2 (Cont.)
FORTRAN IV-PLUS
Generic and Processor-Defined Functions

Function	Definition	Number of Arguments	Generic Name	Type of Argument(s)	Type of Result	PDF Name	Internal Name
Bitwise AND		2	IAND	Integer-2 Integer-4	Integer-2 Integer-4	IIAND JIAND	$IAND $JAND
Bitwise Inclusive OR		2	IOR	Integer-2 Integer-4	Integer-2 Integer-4	IIOR JIOR	$IOR $JOR
Bitwise Exclusive OR		2	IEOR	Integer-2 Integer-4	Integer-2 Integer-4	IIEOR JIEOR	$IEOR $JEOR
Bitwise Complement		1	NOT	Integer-2 Integer-4	Integer-2 Integer-4	INOT JNOT	$INOT $JNOT
Bitwise Shift	a_1 logically shifted left a_2 bits	2	ISHFT	Integer-2 Integer-4	Integer-2 Integer-4	IISHFT JISHFT	$ISHFT $JSHFT

Notes for Table B-2

1. [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x.

2. The remaindering functions of (a_1, a_2) are defined as $a_1 - [a_1/a_2]a_2$, where [x] is defined in 1. above.

3. Functions that cause conversion of an entity from one type to another type provide the same effect as the implied conversion in assignment statements. The functions SNGL with a real argument, and DBLE with a double precision argument, return the value of the argument without conversions.

4. A complex value is expressed as an ordered pair of reals (ar, ai), where ar is the real part and ai is the imaginary part.

5. The argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than zero. The argument of CLOG must not be (0.,0.).

6. The argument of SIN, DSIN, COS, DCOS, TAN, and DTAN must be in radians. These functions use the argument modulo 2 pi.

7. The argument of SQRT and DSQRT must be greater than or equal to zero.

8. The absolute value of the argument of ASIN, DASIN, ACOS, and DACOS must be less than or equal to one.

9. The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

10. The result of ATAN2 and DATAN2 is zero or positive for $a_2 \geq 0$ and negative for $a_1 < 0$. The result is undefined if both arguments are zero.

11. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part is zero, the imaginary part is greater than or equal to zero.

12. The principal value is used for the results of complex functions.

APPENDIX C

FORTRAN PROGRAMMING EXAMPLES

Four examples of FORTRAN programs are given below. These examples are intended to show possible methods of handling input/output, iterative calculations, the FORTRAN Library functions, and subprogram usage in the context of problems likely to face a FORTRAN programmer. These particular programs should not be considered as the correct or optimal approach to the specified problems since many other methods are possible in each case.

The program in example one performs linear regression on a set of X,Y coordinates. The program uses standard formulae to calculate the slope and intercept of the line which best fits the data points entered. The program listing and a sample run follow:

EXAMPLE 1 LISTING:

```
        TYPE 10
5       TYPE 20
        ACCEPT 30,N
        IF (N .LE. 0)   STOP
        TYPE 40,N
        SIGMXY = 0
        SIGMX = 0
        SIGMY = 0
        SIGMXX = 0
        DO 100, J=1,N
                ACCEPT 50,X,Y
                SIGMXY = SIGMXY + X*Y
                SIGMX = SIGMX + X
                SIGMY = SIGMY + Y
                SIGMXX = SIGMXX + X*X
100     CONTINUE
        A = (SIGMXY-SIGMX*SIGMY/N) / (SIGMXX-SIGMX*SIGMX/N)
        B = (SIGMY-A*SIGMX)/N
        TYPE 60,A,B
        GO TO 5
C
C       FORMAT STATEMENTS
C
```

```
10      FORMAT (//' THIS PROGRAM PERFORMS LINEAR REGRESSION'/
        1 ' THE LINE WHICH BEST FITS A SET OF X,Y PAIRS IS CALCULATED.')
20      FORMAT (/' TYPE IN THE NUMBER OF X,Y PAIRS: '$)
30      FORMAT (I2)
40      FORMAT (' TYPE IN ',I2,' LINES OF X,Y PAIRS.'/)
50      FORMAT (2F8.3)
60      FORMAT (/' THE BEST FIT IS Y=',F8.3,' X=',F8.3)
        END
```

EXAMPLE 1 SAMPLE RUN:

```
MCR>RUN EXMPL1$


THIS PROGRAM PERFORMS LINEAR REGRESSION
THE LINE WHICH BEST FITS A SET OF X,Y PAIRS IS CALCULATED.

TYPE IN THE NUMBER OF X,Y PAIRS: 17
TYPE IN 17 LINES OF X,Y PAIRS.

   1.1   4.7
   2.2   9.4
   4.1   9.9
   5.3  14.6
   6.7  23.1
   8.0  29.9
  10.1  37.1
  11.5  41.3
  13.6  54.7
  15.8  59.2
  17.9  64.7
  19.5  71.2
  22.3  79.8
  23.1  81.4
  25.3  88.8
  26.9  91.9
  27.3  95.4

THE BEST FIT IS Y=   3.537 X=   0.340

TYPE IN THE NUMBER OF X,Y PAIRS: 6
TYPE IN  6 LINES OF X,Y PAIRS.

  0.41  -4.3
  0.99  -7.2
  0.31  -3.6
  0.02  -0.7
  0.76  -5.9
  0.43  -4.4

THE BEST FIT IS Y=  -6.304 X=  -1.282

TYPE IN THE NUMBER OF X,Y PAIRS: 0
EXMPL1  --  STOP
```

The program in example two manipulates data representing test scores. The scores are read from the source file, placed in descending order, and sent to an output file. Then the absolute total and histogram of the test scores in each 10-point interval are output on the terminal. The program listing and a sample run follow:

EXAMPLE 2 LISTING:

```
      LOGICAL*1 STARS(80)
      INTEGER ARRAY(200),HIST(10)
      DATA STARS /80*'*'/
      DATA HIST / 10*0/
      DO 100, I=1,200
          READ(1,20,END=105) ARRAY(I)
100   CONTINUE
105   ISIZE = I-1
      DO 120, J=1,ISIZE-1
          DO 110, K=J+1,ISIZE
              IF(ARRAY(J).GE.ARRAY(K)) GO TO 110
              ITMP = ARRAY(J)
              ARRAY(J) = ARRAY(K)
              ARRAY(K) = ITMP
110       CONTINUE
120   CONTINUE
      DO 125, K=1,ISIZE
          WRITE(2,20) ARRAY(K)
125   CONTINUE
      DO 130, K=1,ISIZE
          N = (MIN0(ARRAY(K),99) - 1) / 10 + 1
          HIST(N) = HIST(N) + 1
130   CONTINUE
      TYPE 35
      DO 150, K=10,100,10
          TYPE 40, HIST(K/10), K-9, K
          IF(HIST(K/10) .EQ. 0) GO TO 150
          TYPE 45, (STARS(M), M=1,HIST(K/10))
150   CONTINUE
      TYPE 50, ISIZE
      STOP
C
C     FORMAT STATEMENTS
C
20    FORMAT(I3)
35    FORMAT(//1X,'The number of test scores and a histogram'/
     1 '  in each 10 point interval follows:'/)
40    FORMAT(/1X,I3,' in the range ',I3,' to ',i3,$)
45    FORMAT(1H+,2X,80A1)
50    FORMAT(//' The total number of test scores = ',I3)
      END
```

EXAMPLE 2 SAMPLE RUN:

```
MCR>RUN EXMPL2$


The number of test scores and a histogram
 in each 10 point interval follows:

  1 in the range   1 to  10  *
  2 in the range  11 to  20  **
  2 in the range  21 to  30  **
 10 in the range  31 to  40  **********
 13 in the range  41 to  50  *************
 11 in the range  51 to  60  ***********
 19 in the range  61 to  70  *******************
 35 in the range  71 to  80  ***********************************
 40 in the range  81 to  90  ****************************************
 17 in the range  91 to 100  *****************

The total number of test scores = 150
```

Example three shows a method of calculating the prime factors of an integer. A simple table look-up method was used to determine the necessary primes. Note the unusual use of FORTRAN carriage control to facilitate the prime factor output. MOD is a Library function and is described in Appendix B. The program listing and a sample run follow:

EXAMPLE 3 LISTING:

```
        INTEGER P,HOLD
        TYPE 50
80      TYPE 100
        ACCEPT 105,NUMBER
        IF (NUMBER .LE. 0)      STOP
        TYPE 110
        ISQRT = SQRT(FLOAT(NUMBER))
        P = 1
        IFLAG = 0
        HOLD = NUMBER
        IF (HOLD .LE. 3) GO TO 240
200     P = NPRIME(P)
205     IREM = MOD(HOLD,P)
        IF (IREM .EQ. 0) GO TO 400
        IF (P .LE. ISQRT) GO TO 200
        IF (IFLAG .NE. 0) GO TO 300
240     TYPE 250,NUMBER
        GO TO 80
```

```
300     IF (HOLD .GT. 1) TYPE 350,HOLD
        GO TO 80
400     IFLAG = 1
        HOLD = HOLD/P
        IF (HOLD .EQ. 1) GO TO 500
        TYPE 450,P
        GO TO 205
500     TYPE 350,P
        GO TO 80
C
C       FORMAT STATEMENTS
C
50      FORMAT (//1X,'THIS IS A PROGRAM TO FIND THE PRIME FACTORS OF',
        1  ' AN INTEGER < 32767.'//' ENTERING A NEGATIVE OR ZERO',
        2  ' NUMBER TERMINATES EXECUTION.'/)
100     FORMAT (//' ENTER #  ',$)
105     FORMAT (I5)
110     FORMAT (/)
250     FORMAT (1H+,I5,' IS A PRIME NUMBER')
350     FORMAT (1H+,I5,$)
450     FORMAT (1H+,I5,'*',$)
        END

        FUNCTION NPRIME(OLD)
        DIMENSION MPRIME(46)
        INTEGER OLD
        DATA MPRIME/2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
        1  53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,
        2  127,131,137,139,149,151,157,163,167,173,179,181,
        3  191,193,197,199/
        IF (OLD .EQ. 1) N = 0
        N = N + 1
        NPRIME = MPRIME(N)
        RETURN
        END
```

EXAMPLE 3 SAMPLE RUN:

```
MCR>RUN EXMPL3$

THIS IS A PROGRAM TO FIND THE PRIME FACTORS OF AN INTEGER < 32767.
ENTERING A NEGATIVE OR ZERO NUMBER TERMINATES EXECUTION.


ENTER #  3983

    7*  569

ENTER #  32761

  181*  181

ENTER #  32749

32749 IS A PRIME NUMBER
```

```
ENTER #  8192

    2*    2*    2*    2*    2*    2*    2*    2*    2*    2*    2*    2*    2

ENTER #  32751

    3*    3*    3* 1213

ENTER #  32767

    7*   31*  151


ENTER #  4099

 4099 IS A PRIME NUMBER


ENTER #  0
EXMPL3  --  STOP
```

Example four demonstrates a simple way to generate random numbers in a given range using the FORTRAN Library function RAN. A program listing and sample run follow:

EXAMPLE 4 LISTING:

```
        REAL MAX,MIN
        TYPE 10
        TYPE 20
        ACCEPT 30,J
        TYPE 40
        ACCEPT 60, MIN
        TYPE 50
        ACCEPT 60, MAX
        L = 0
        M = 0
        DO 100, K=1,J
                X = RAN(L,M) * (MAX-MIN) + MIN
                TYPE 60, X
                WRITE (2,60) X
100     CONTINUE
        STOP
C
C       FORMAT STATEMENTS
C
10      FORMAT(//1X,'THIS IS A PROGRAM TO GENERATE A FILE OF',
        1   ' RANDOM #''S IN A GIVEN RANGE.'/)
20      FORMAT(1X,'ENTER THE NUMBER OF RANDOM #''S TO GENERATE:',$)
30      FORMAT(I3)
40      FORMAT(1X,'ENTER THE MINIMUM VALUE:',$)
50      FORMAT(1X,'ENTER THE MAXIMUM VALUE:',$)
60      FORMAT(F10.4)
        END
```

EXAMPLE 4 SAMPLE RUN:

```
MCR>RUN EXMPL4$

THIS IS A PROGRAM TO GENERATE A FILE OF RANDOM #'S IN A GIVEN RANGE.

ENTER THE NUMBER OF RANDOM #'S TO GENERATE:40
ENTER THE MINIMUM VALUE:-5.0
ENTER THE MAXIMUM VALUE:5.0
  -4.8465
  -4.5386
  -3.6130
  -0.8307
  -2.4673
   2.6721
  -1.7612
  -4.6161
  -1.8459
   0.4690
  -0.5726
   2.3432
  -0.7876
   4.1861
   2.2046
  -4.4474
   3.4746
   0.8739
   3.9721
  -4.0326
   0.0559
  -3.3713
  -0.7312
  -4.0458
   2.3063
   0.2498
   0.7423
   2.2060
  -3.4453
  -0.5254
  -2.1446
   1.8604
   0.4644
  -3.9575
   2.0757
  -1.9288
  -0.2538
  -4.1638
  -2.6987
   1.2823
 EXMPL4  --  STOP
```

APPENDIX D

VIRTUAL ARRAYS

D.1 INTRODUCTION TO VIRTUAL ARRAYS

User programs executing on a PDP-11 family computer can directly address at most 65,536 bytes (64K bytes) of main memory at any instant. This main memory can contain a mixture of FORTRAN compiled code, variables and arrays, and external procedures. Program overlays permit larger programs to be executed in the directly-addressable main memory. Virtual arrays provide another mechanism for processing large arrays that would otherwise exceed the 64K byte limit on directly-addressable memory.

A virtual array is an array whose storage is allocated in physical main memory outside of the program's directly-addressable main memory. The use of virtual arrays in a program frees directly-addressable memory for executable code and other data storage.

NOTE

Virtual arrays are not supported in all PDP-11 FORTRAN implementations. Virtual arrays are available in Version 2 of the FORTRAN IV compiler for the RSX-11M and RT-11 operating systems.

D.2 VIRTUAL STATEMENT

The VIRTUAL statement defines the number of dimensions in a virtual array and the number of elements in each dimension. The VIRTUAL statement is similar in form and usage to the DIMENSION statement (Section 7.3).

The VIRTUAL statement has the form:

```
VIRTUAL a(d) [, a(d)]...
```

a is the symbolic name of an array.
d is a dimension declarator.

Each a(d) is an array declarator as described in Section 2.6.1.

The VIRTUAL statement allocates a number of (virtual) storage locations to each array named in the statement. For further information concerning arrays and the storage of array elements, see Section 2.6.

The data type of a virtual array is specified in the same way as the data type of a variable or an array; that is, implicitly by the

initial letter of the name, or explicitly by a type declaration statement.

The name of a virtual array can appear in a type declaration statement; however, in each program unit, an array name can appear in only one array declarator. The name of a virtual array cannot appear in a COMMON statement, EQUIVALENCE statement, or DATA statement.

VIRTUAL statements must not be labeled.

Examples

```
VIRTUAL    A(100), B(100,100)
VIRTUAL    C(4,4,4,4,4)
```

D.3 SIZE OF VIRTUAL ARRAYS

The VIRTUAL statement allocates a number of storage locations for each element in each dimension of the array. Each storage location is one, two, four or eight bytes in length, as determined by the data type of the array. The total number of elements in the array is equal to the product of all dimension declarators in the array declarator.

A VIRTUAL array can have a maximum of 32767 elements; each element requires from one to eight bytes of storage. Thus a maximum size LOGICAL*1 virtual array requires 32767 bytes of physical main memory; a maximum size REAL*8 virtual array requires 262,136 bytes of physical main memory.

The declaration of a virtual array does not significantly reduce directly-addressable memory available to the program. VIRTUAL arrays are constrained, however, by the total amount of physical main memory available in the system. If adequate memory is not available to contain all virtual arrays declared in a program, the program cannot be executed. Refer to the appropriate FORTRAN User's Guide for additional information on size constraints.

D.4 RESTRICTIONS ON VIRTUAL ARRAY USAGE

The names of virtual arrays and virtual array elements must not be used in some contexts:

1. A virtual array name must not be used in a COMMON statement (Section 7.4).

2. The name of a virtual array or virtual array element must not be used in an EQUIVALENCE statement (Section 7.5).

3. A virtual array or virtual array element cannot be assigned an initial value by a DATA statement (Section 7.7).

4. Virtual arrays cannot be used to contain object time format specifications (Section 6.6). The name of a virtual array or virtual array element must not appear as a format specifier in a READ, WRITE, PRINT, TYPE, ACCEPT, ENCODE or DECODE statement.

5. The name of a virtual array or virtual array element must not be specified as the buffer argument (third argument inside parentheses) of an ENCODE or DECODE statement (Section 5.10).

6. The name of a virtual array element must not be used as an actual argument to a subprogram if the subprogram assigns a value to the corresponding dummy argument.

Examples

Valid Usage

```
      VIRTUAL A(1000), B(2000)
      READ(1,*) A
      DO 10,I=1,1000
10    B(I)=-A(I)*2
      WRITE(2,*) (A(I),I=1,1000)
      CALL ABC(A,B)
```

Invalid Usage

```
DIMENSION B(10)
VIRTUAL A(10)
DATA  A(1)/2.5/                    (Used in DATA statement)
COMMON /X/  A                      (Used in COMMON statement)
EQUIVALENCE  (B,A)                 (Used in EQUIVALENCE statement)
EQUIVALENCE (A(1),X)               (Used in EQUIVALENCE statement)
WRITE(1,A) (B(I),I=1,10)           (Used as format specifier)
ENCODE(4,100,A(3)) X,Y             (Used as ENCODE output buffer)
```

D.5 VIRTUAL ARRAY REFERENCES IN SUBPROGRAMS

Actual and dummy arguments become associated at the time control is tranferred to a subprogram (Section 8.1). Actual and dummy arguments that become associated must agree in data type. A dummy argument declared as a non-virtual array can only become associated with an actual argument that is a non-virtual array or array element. A dummy argument declared as a virtual array can only become associated with an actual argument that is also the name of a virtual array.

An actual argument which is a reference to a virtual array element can only become associated with a dummy argument which is declared as a simple variable (Section 2.2). In effect, an actual argument that is a virtual array element is treated as an expression. A value must have been assigned to the element before it is used as an actual argument. The subprogram must not alter the value of the corresponding dummy argument. Note that neither of these restrictions applies to the use of a non-virtual array element as an actual argument.

Examples

Valid Usage

```
      VIRTUAL A(1000),B(1000)
      ...
      WRITE(5,*) SCALE(A,1000,B(3))
      END

      REAL FUNCTION SCALE (X,N,W)
      VIRTUAL X(N)
      S=0.
      DO 10, I=1,N
 10   S=S+X(I)*W
      SCALE=S
      END
```

Invalid Usage

```
      VIRTUAL A(1000)
      REAL    B(4000)
      CALL    ABC(A, B, A(3))
      END

      SUBROUTINE ABC(X,Y,Z)
      REAL X(1000)                (Invalid;  actual argument is virtual)
      VIRTUAL Y(4000)             (Invalid;     actual     argument   is
                                  non-virtual)
      Z=2.3                       (Invalid;  actual argument is virtual
                                  array element)
      END
```

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

PDP-11 FORTRAN
Language Reference
Manual
DEC-11-LFLRA-C-DN1

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

Please cut along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754